# Software System Reliability and Security

Edited by
Manfred Broy
Johannes Grünbauer
Tony Hoare

IOS Press

*This publication is supported by:* The NATO Programme for Security through Science

SOFTWARE SYSTEM RELIABILITY AND SECURITY

**NATO Security through Science Series**

This Series presents the results of scientific meetings supported under the NATO Programme for Security through Science (STS).

Meetings supported by the NATO STS Programme are in security-related priority areas of Defence Against Terrorism or Countering Other Threats to Security. The types of meeting supported are generally "Advanced Study Institutes" and "Advanced Research Workshops". The NATO STS Series collects together the results of these meetings. The meetings are co-organized by scientists from NATO countries and scientists from NATO's "Partner" or "Mediterranean Dialogue" countries. The observations and recommendations made at the meetings, as well as the contents of the volumes in the Series, reflect those of participants and contributors only; they should not necessarily be regarded as reflecting NATO views or policy.

**Advanced Study Institutes** (ASI) are high-level tutorial courses to convey the latest developments in a subject to an advanced-level audience.

**Advanced Research Workshops** (ARW) are expert meetings where an intense but informal exchange of views at the frontiers of a subject aims at identifying directions for future action.

Following a transformation of the programme in 2004 the Series has been re-named and re-organised. Recent volumes on topics not related to security, which result from meetings supported under the programme earlier, may be found in the NATO Science Series.

The Series is published by IOS Press, Amsterdam, and Springer Science and Business Media, Dordrecht, in conjunction with the NATO Public Diplomacy Division.

**Sub-Series**

| | | |
|---|---|---|
| A. | Chemistry and Biology | Springer Science and Business Media |
| B. | Physics and Biophysics | Springer Science and Business Media |
| C. | Environmental Security | Springer Science and Business Media |
| D. | Information and Communication Security | IOS Press |
| E. | Human and Societal Dynamics | IOS Press |

http://www.nato.int/science
http://www.springer.com
http://www.iospress.nl

Sub-Series D: Information and Communication Security – Vol. 9 ISSN: 1574-5589

# Software System Reliability and Security

Edited by

Manfred Broy

*Technische Universität München, Germany*

Johannes Grünbauer

*Technische Universität München, Germany*

and

Tony Hoare

*Microsoft Research, UK*

IOS Press

Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

Published in cooperation with NATO Public Diplomacy Division

Proceedings of the NATO Advanced Research Institute on Software System Reliability and Security
Marktoberdorf, Germany
1–13 August 2006

© 2007 IOS Press.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without prior written permission from the publisher.

ISBN 978-1-58603-731-4
Library of Congress Control Number: 2007922976

*Publisher*
IOS Press
Nieuwe Hemweg 6B
1013 BG Amsterdam
Netherlands
fax: +31 20 687 0019
e-mail: order@iospress.nl

*Distributor in the UK and Ireland*
Gazelle Books Services Ltd.
White Cross Mills
Hightown
Lancaster LA1 4XS
United Kingdom
fax: +44 1524 63232
e-mail: sales@gazellebooks.co.uk

*Distributor in the USA and Canada*
IOS Press, Inc.
4502 Rachael Manor Drive
Fairfax, VA 22032
USA
fax: +1 703 323 3668
e-mail: iosbooks@iospress.com

LEGAL NOTICE
The publisher is not responsible for the use which might be made of the following information.

PRINTED IN THE NETHERLANDS

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Preface

Today almost every complex technical system used in industry, science, commerce and communication is more or less interfaced with software and software systems. This dictates that most information exchange is closely related to software and computer systems. The consequence of this wide distribution of software is a high dependency on its functioning and quality. Because of this dependency and distribution, making information systems safe, reliable, as well as secure and protecting information against all kinds of attack is an essential research topic, particularly in computer science.

Scientific foundations have been developed for programming and building computer systems. These foundations cover a broad spectrum of issues and work with formal models and description techniques in order to support a deep and precise understanding and managing of a system's properties and interplay. In addition, software engineering has many additional applications, ranging from telecommunications to embedded systems. For example software engineering has now become essential in automotive and aircraft industry, and has been intergral in furthering computer networks distributed over wide-area networks. A vast proportion of information exchange is influenced by computer systems and information security is important for reliable and secure software and computer systems.

Information security covers the protection of information against unauthorized disclosure, transfer, modification, and destruction, whether accidentally or intentionally. Attacks against computer systems can cause considerable economic and physical damage. Quality of life in general and of individual citizens, and the effectiveness of the economy critically depends on our ability to build software in a transparent and efficient way. Furthermore, we must be able to enhance the software development process systematically in order to ensure safety, security and reliability. This, in turn, requires very high software reliability, i. e., an extremely high confidence in the ability of the software to perform flawlessly. The foundations of software technology provide models that enable us to capture application domains and their requirements, but also to understand the structure and working of software systems, software architectures and programs. New developments must pay due diligence to the importance of security-related aspects, and align current methods and techniques to information security, integrity, and system reliability. However, based on the specific needs in applications of software technology, models and formal methods must serve the needs and the quality of advanced software engineering methods, especially taking into account security aspects in Information Technology.

As a consequence of the wide distribution of software and software infrastructure, information security depends on the quality and excellent understanding of its functioning. Only when this functionality is guaranteed as safe, customers, and information are protected against adversarial attacks. Thus, to make communication and computation secure against catastrophic failure and malicious interference, it is essential to build secure software systems and methods for their development. Such development is difficult, mainly because of the conflict between development costs and verifiable correctness.

In the summer of 2006, a group of internationally renowned researchers in computer science met and lectured on the topics described above. The articles in this book describe the state-of-the-art ideas on how to meet these challenges in software engineering.

*Rajeev Alur* describes the foundations of model checking of programs with finite data and stack-based control flow. *Manfred Broy* introduces an abstract theory for systems, components, composition, architectures, interfaces, and compatibility. In his article he applies this theory to object orientation and elaborates on the application of that theory covering notions for a formal model of objects, classes, components, and architectures as well as those of interfaces of classes and components and their specification. *Ernie Cohen* explains how to use ordinary program invariants to prove properties of cryptographic protocols.

Networked computer systems face a range of threats from hostile parties on the network leading to violations of design goals such as confidentiality, privacy, authentication, access control, and availability. The purpose of *Andrew Gordon*'s article is to introduce an approach to this problem based on process calculi. Transactions are the essential components of electronic business systems, and their safety and security are of increasing concern. *Tony Hoare* presents a theoretical model of compensable transactions, showing how long running transactions may be correctly composed out of shorter ones. *Orna Kupferman* presents on "Applications of Automata-Theory in Formal Verification". In this automata-theoretic approach to verification, she reduces questions about programs and their specifications to questions about automata.

In a distributed system with no central management such as the Internet, security requires a knowledge about who can be trusted for each step in establishing it, and why. *Butler W. Lampson* explains the "speaks for" relation between principals describing how authority is delegated. *Axel van Lamsweerde* contributes model-based requirements engineering. Models for agents, operations, obstacles to goals, and security threats are introduced and a model building with the KAOS method is presented. *Wolfgang Paul* outlines a correctness proof for a distributed real time system – for the first time in a single place – from the gate level to the computational model of a CASE tool.

*Amir Pnueli* describes an approach for the synthesis of (hardware and software) designs from LTL specifications. This approach is based on modelling the synthesis problem which is similar to the problem of finding a winning strategy in a two-person game. *K. Venkatesh Prasad* introduces the notion of a "mobile networked embedded system", in which a mobile entity is composed of internally and externally networked software components. He discusses the challenges related to designing a mobile networked embedded system with regards to security, privacy, usability, and reliability. Finally, *Wolfram Schulte* explains the "Spec# Approach", which provides method contracts in the form of pre- and post-conditions as well as object invariants. He describes the design of *Spec#*'s state-of-the-art program verifier for object-oriented programs.

The contributions in this volume have emerged from lectures of the 27th International Summer School on *Software System Reliability and Security*, held at Marktoberdorf from August 1 to August 13, 2006. More than 100 participants from 28 countries attended, including students, lecturers and staff. The Summer School provided two weeks of learning, discussion and development of new ideas, and was a fruitful event, at both the professional and social level.

We would like to thank all lecturers, staff, and hosts in Marktoberdorf. In particular special thanks goes to our secretaries Dr. Katharina Spies, Silke Müller, and Sonja Werner for their great and gentle support.

The Marktoberdorf Summer School was arranged as an Advanced Study Institute of the *NATO Security Through Science Programme* with support from the town and county of Marktoberdorf and the *Deutscher Akademischer Austausch Dienst (DAAD)*. We thank all authorities involved.

THE EDITORS

# Contents

Preface v

Logics and Automata for Software Model-Checking 1
*Rajeev Alur and Swarat Chaudhuri*

Specifying, Relating and Composing Object Oriented Interfaces, Components and Architectures 22
*Manfred Broy*

Using Invariants to Reason About Cryptographic Protocols 73
*Ernie Cohen*

Verified Interoperable Implementations of Security Protocols 87
*Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon and Stephen Tse*

Compensable Transactions 116
*Tony Hoare*

Automata on Infinite Words and Their Applications in Formal Verification 135
*Orna Kupferman*

Practical Principles for Computer Security 151
*Butler Lampson*

Engineering Requirements for System Reliability and Security 196
*Axel van Lamsweerde*

Pervasive Verification of Distributed Real-Time Systems 239
*Steffen Knapp and Wolfgang Paul*

Verification and Synthesis of Reactive Programs 298
*Amir Pnueli*

Security, Privacy, Usability and Reliability (SPUR) in Mobile Networked Embedded Systems: The Case of Modern Automobiles 341
*K. Venkatesh Prasad and T.J. Giuli*

A Verifying Compiler for a Multi-Threaded Object-Oriented Language 351
*K. Rustan, M. Leino and Wolfram Schulte*

Author Index 417

This page intentionally left blank

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Logics and Automata for Software Model-Checking [1]

Rajeev ALUR and Swarat CHAUDHURI
*University of Pennsylvania*

**Abstract.** While model-checking of pushdown models is by now an established technique in software verification, temporal logics and automata traditionally used in this area are unattractive on two counts. First, logics and automata traditionally used in model-checking cannot express requirements such as pre/post-conditions that are basic to software analysis. Second, unlike in the finite-state world, where the $\mu$-calculus has a symbolic model-checking algorithm and serves as an "assembly language" of temporal logics, there is no unified formalism to model-check linear and branching requirements on pushdown models. In this survey, we discuss a recently-proposed re-phrasing of the model-checking problem for pushdown models that addresses these issues. The key idea is to view a program as a generator of structures known as *nested words* and *nested trees* (respectively in the linear and branching-time cases) as opposed to words and trees. Automata and temporal logics accepting languages of these structures are now defined, and linear and branching time model-checking phrased as language inclusion and membership problems for these languages. We discuss two of these formalisms—automata on nested words and a fixpoint calculus on nested trees—in detail. While these formalisms allow a new frontier of program specifications, their model-checking problem has the same worst-case complexity as their traditional analogs, and can be solved symbolically using a fixpoint computation that generalizes, and includes as a special case, "summary"-based computations traditionally used in interprocedural program analysis.

**Keywords.** Temporal and fixpoint logics, Automata, Software model-checking, Verification, Program analysis

## 1. Introduction

Because of concerted research over the last twenty-five years, model-checking of reactive systems is now well-understood theoretically as well as applied in practice. The theories of temporal logics and automata have played a foundational role in this area. For example, in linear-time model-checking, we are interested in questions such as: "do all traces of a protocol satisfy a certain safety property?" This question is phrased language-theoretically as: is the set of all possible system traces included in the language of safe behaviors? In the world of finite-state reactive programs, both these languages are $\omega$-regular [22]. On the other hand, in branching-time model-checking, the specification de-

[1]This research was partially supported by ARO URI award DAAD19-01-1-0473 and NSF award CPA 0541149.

fines an $\omega$-regular language of trees, and the model-checking problem is to determine if the tree unfolding of the system belongs to this language [17].

Verification of software is a different ball-game. Software written in a modern programming language has many features such as the stack, the heap, and concurrent execution. Reasoning about these features in any automated manner is a challenge—finding ways to model-check them is far harder. The approach that software model-checking takes [10] is that of *data abstraction*: finitely approximate the data in the program, but model the semantics of procedure calls and returns precisely. The chosen abstractions are, thus, *pushdown models* or finite-state machines equipped with a pushdown stack (variants such as *recursive state machines* [1] and boolean programs [9] have also been considered). Such a machine is now viewed as a generator of traces or trees modeling program executions or the program unfolding.

There are, of course, deviations from the classical setting: since pushdown models have unbounded stacks and therefore infinitely many configurations, answering these queries requires *infinite-state model-checking*. Many positive results are known in this area—for instance, model-checking the $\mu$-calculus, often called the "assembly language for temporal logics," is decidable on sequential pushdown models [24,12]. However, many attractive computational properties that hold in the finite-state world are lost. For instance, consider the reachability property: "a state satisfying a proposition $p$ is reachable from the current state," expressible in the $\mu$-calculus by a formula $\varphi = \mu X.(p \vee \langle\rangle X)$. In finite-state model checking, $\varphi$ not only states a property, but syntactically encodes a symbolic fixpoint computation: start with the states satisfying $p$, add states that can reach the previous states in one step, then two steps, and so on. This is the reason why hardware model-checkers like SMV translate a specification given in a simpler logic into the $\mu$-calculus, which is now used as a directive for fixpoint computation. Known model-checking algorithms for the $\mu$-calculus on pushdown models, however, are complex and do not follow from the semantics of the formula. In particular, they cannot capture the natural, "summarization"-based fixpoint computations for interprocedural software analysis that have been known for years [19,21].

Another issue with directly applying classical temporal specifications in this context is expressiveness. Traditional logics and automata used in model-checking define *regular* languages of words and trees, and cannot argue about the balanced-parenthesis structure of calls and returns. Suppose we are now interested in *local reachability* rather than reachability: "a state satisfying $p$ is reachable *in the same procedural context* (i.e., before control returns from the current context, and not within the scope of new contexts transitively spawned from this context via calls)." This property cannot be captured by regular languages of words or trees. Other requirements include Hoare-Floyd-style preconditions and postconditions [16] ("if $p$ holds at a procedure call, then $q$ holds on return"), interface contracts used in real-life specification languages such as JML [11] and SAL [14], stack-sensitive access control requirements arising in software security [23], and interprocedural dataflow analysis [18].

While checking pushdown requirements on pushdown models is undecidable in general, individual static analysis techniques are available for all the above applications. There are practical static checkers for interface specification languages and stack inspection-type properties, and interprocedural dataflow analysis [19] can compute dataflow information involving local variables. Their foundations, unfortunately, are not properly understood. What class of languages do these properties correspond to? Can

we offer the programmer flexible, decidable temporal logics or automata to write these requirements? These are not merely academic questions. A key practical attraction of model-checking is that a programmer, once offered a temporal specification language, can tailor a program's requirements without getting lost in implementation details. A logic as above would extend this paradigm to interprocedural reasoning. Adding syntactic sugar to it, one could obtain domain-specific applications—for example, one can conceive of a language for module contracts or security policies built on top of such a formalism.

In this paper, we summarize some recent work on software model-checking [4,6,7,2, 3] that offers more insights into these issues by re-phrasing the model-checking problem for sequential pushdown models. In classical linear-time model-checking, the problem is to determine whether the set of linear behaviors of a program abstraction is included in the set of behaviors satisfying the specification. In branching-time model-checking, the question is whether the tree unfolding of the program belongs to the language of trees satisfying the requirement. In other words, a program model is viewed as a generator of a word or tree structure. In the new approach, programs are modeled by pushdown models called *nested state machines*, whose executions and unfoldings are given by graphs called *nested words* and *nested trees*. More precisely, a nested word is obtained by augmenting a word with a set of extra edges, known as *jump-edges*, that connect a position where a call happens to its matching return. As calls and returns in program executions are properly nested, jump-edges never cross. To get a nested tree, we add a set of jump-edges to a tree. As a call may have a number of matching returns along the different paths from it, a node may now have multiple outgoing jump-edges. Temporal logics and finite-state automata accepting languages of nested words and trees are now defined. The linear-time model-checking question then becomes: is the set of *nested words* modeling executions of a program included in the set of nested words accepted by the specification formula/automaton? For branching-time model-checking, we ask: does the *nested tree* generated by a program belong to a specified language of nested trees? It turns out that this tweak makes a major difference computationally as well as in expressiveness.

Let us first see what an automaton on nested words (NWA) [6,7] would look like. Recall that in a finite automaton, the state of the automaton at a position depends on the state and the input symbol at the preceding position. In a nested word, a "return" position has two incoming edges—one from the preceding time point, and one from the "matching call." Accordingly, the state of an NWA may depend on the states of the automaton at both these points. To see how this would help, consider the local reachability property. Consider an automaton following a program execution, and suppose it is in a state $q$ that states that a target state has not yet been seen. Now suppose it encounters a procedure call. A finite automaton on words would follow the execution into the new procedural context and eventually "forget" the current state. However, an NWA can retrieve the state $q$ once the execution returns from this call, and continue to search only the current context. Properties such as this can also be expressed using temporal logics on nested words [4,13], though we will not discuss the latter in detail in this survey.

For branching-time model-checking, we use a fixpoint calculus called NT-$\mu$ [2]. The variables of the calculus evaluate not over sets of states, but rather over sets of substructures that capture *summaries* of computations in the "current" program block. The fixpoint operators in the logic then compute fixpoints of summaries. For a node $s$ of a nested tree representing a call, consider the tree rooted at $s$ such that the leaves correspond to

exits from the current context. In order to be able to relate paths in this subtree to the trees rooted at the leaves, we allow marking of the leaves: a 1-ary summary is specified by the root $s$ and a subset $U$ of the leaves of the subtree rooted at $s$. Each formula of the logic is evaluated over such a summary. The central construct of the logic corresponds to concatenation of call trees: the formula $\langle call \rangle \varphi\{\psi\}$ holds at a summary $\langle s, U \rangle$ if the node $s$ represents a "call" to a new context starting with node $t$, there exists a summary $\langle t, V \rangle$ satisfying $\varphi$, and for each leaf $v$ that belongs to $V$, the subtree $\langle v, U \rangle$ satisfies $\psi$. Intuitively, a formula $\langle call \rangle \varphi\{\psi\}$ asserts a constraint $\varphi$ on the new context, and requires $\psi$ to hold at a designated set of return points of this context. To state local reachability, we would ask, using the formula $\varphi$, that control returns to the current context, and, using $\psi$, that the local reachability property holds at some return point. While this requirement seems self-referential, it may be captured using a fixpoint formula.

It turns out that NWAs and NT-$\mu$ have numerous attractive properties. For instance, NWAs have similar closure properties such as regular languages, easily solvable decision problems, Myhill-Nerode-style characterizations, etc. NT-$\mu$ can express all properties expressible in the $\mu$-calculus or using NWAs, and has a characterization in terms of *automata on nested trees* [3]. In this survey, we focus more on the computational aspects as applicable to program verification, in particular the model-checking problem for NT-$\mu$. The reason is that NT-$\mu$ can capture both linear and branching requirements and, in spite of its expressiveness, can be model-checked efficiently. In fact, the complexity of its model-checking problem on pushdown models (EXPTIME-complete) is the same as that of far weaker logics such as CTL or the alternation-free $\mu$-calculus. Moreover, just as formulas of the $\mu$-calculus syntactically encode a terminating, symbolic fixpoint computation on finite-state systems, formulas of NT-$\mu$ describe a directly implementable symbolic model-checking algorithm. In fact, this fixpoint computation generalizes the kind of summary computation traditionally known in interprocedural program analysis, so that, just like the $\mu$-calculus in case of finite-state programs, NT-$\mu$ can arguably be used as an "assembly language" for interprocedural computations.

The structure of this paper is as follows. In Sec. 2, we define nested words and trees, and introduce nested state machines as abstractions of structured programs. In Sec. 3, we define specifications on nested structures, studying NWAs and NT-$\mu$ in some detail. In Sec. 4, we discuss in detail the symbolic model-checking algorithm for NT-$\mu$.

## 2. Models

A typical approach to software model-checking uses *data abstraction*, where the data in a structured program is abstracted using a finite set of boolean variables that stand for predicates on the data-space [8,15]. The resulting models have finite-state but stack-based control flow. In this section, we define nested state machines, one such model. The behaviors of these machines are modeled by nested words and trees, the structures on which our specifications are interpreted.

As a running example, in the rest of this paper, we use the recursive procedure *foo*. The procedure may read or write a global variable $x$ or perform an action *think*, has non-deterministic choice, and can call itself recursively. Actions of the program are marked by labels L1–L5 for easy reference. We will abstract this program and its behaviors, and subsequently specify it using temporal logics and automata.

### 2.1. Nested words

*Nested words* form a class of directed acyclic graphs suitable for abstracting executions of structured programs. In this application, a nested word carries information about a sequence of program states as well as the nesting of procedure calls and returns during the execution. This is done by adding to a word a set of extra edges, known as *jump-edges*, connecting positions where calls happen to their matching returns. Because of the balanced-parentheses semantics of calls and returns, jump-edges are properly nested.

Formally, let $\Sigma$ be a finite alphabet. Let a finite word $w$ of length $n$ over $\Sigma$ be a map $w : \{0, 1, \ldots, n-1\} \rightarrow \Sigma$, and an infinite word be a map $w : \mathbb{N} \rightarrow \Sigma$. We sometimes view a word as a graph with positions as nodes, and edges $(i, j)$ connecting successive positions $i, j$. A *nested word* over $\Sigma$ is a pair $\mathcal{W} = (w, \hookrightarrow)$, where $w$ is a finite or infinite word over $\Sigma$, and $\hookrightarrow \subseteq \mathbb{N} \times (\mathbb{N} \cup \infty)$ is a set of *jump-edges*. A position $i$ in a nested word such that $i \hookrightarrow \infty$ or $i \hookrightarrow j$ for some $j$ is called a *call position*, and a position $j$ such that $i \hookrightarrow j$ for some $i$ is called *return position*). The remaining positions are said to be *local*. The idea is that if a jump-edge $(i, j)$ exists, then position $j$ is the matching return of a call at position $i$; a jump-edge $(i, \infty)$ implies that there is a call at position $i$ that never returns. The jump-edge relation must satisfy the following conditions:

1. if $i \hookrightarrow j$, then $i < j - 1$ (in other words, a jump-edge is a non-trivial forward jump in the word);
2. for each $i$, there is at most one $x \in \mathbb{N} \cup \{\infty\}$ such that $i \hookrightarrow x$ or $x \hookrightarrow i$ (a call either never returns or has a unique matching return, and a return has a unique matching call);
3. if $i \hookrightarrow j$ and $i' \hookrightarrow j'$ and $i < i'$, then either $j < i'$ or $j' < j$ (jump-edges are properly nested);
4. if $i \hookrightarrow \infty$, then for all calls $i' < i$, either $i' \hookrightarrow \infty$ or $i' \hookrightarrow j$ for some $j < i$ (if a call never returns, neither do the calls that are on the stack when it is invoked).

If $i \hookrightarrow j$, then we call $i$ the *jump-predecessor* of $j$ and $j$ the *jump-successor* of $i$. Let an edge $(i, i+1)$ in $w$ be called a call and return edge respectively if $i$ is a call and $(i+1)$ is a return, and let all other edges be called *local*.

Let us now turn to our running example. We will model an execution by a nested word over an alphabet $\Sigma$. The choice of $\Sigma$ depends on the desired level of detail—we pick the symbols *wr*, *rd*, *en*, *ex*, *tk*, and *end*, respectively encoding *write(x)*, *read(x)*, a procedure call leading to a beginning of a new context, the return point once a context ends, the statement *think*, and the statement *return*. Now consider the execution where *foo* calls itself twice recursively, then executes *think*, then returns once, then loops infinitely. The word encoding this execution is $w = wr.en.wr.en.wr.tk.rd.ex.(rd)^\omega$. A prefix of the nested word is shown in Fig. 1-(a). The jump-edges are dashed, and call, return and local positions are drawn in different styles. Note the jump-edge capturing the call that never returns.

Now we show a way to encode a nested word using a word. Let us fix a set of *tags* $I = \{call, ret, loc\}$. The *tagged word* $Tag(\mathcal{W})$ of a nested word $\mathcal{W} = (w, \hookrightarrow)$ over $\Sigma$ is obtained by labeling the edges in $w$ with tags indicating their types. Formally, $Tag(\mathcal{W})$ is a pair $(w, \eta)$, where $\eta$ is a map labeling each edge $(i, i+1)$ such that $\eta(i, i+1)$ equals *call* if $i$ is a call, *ret* if $(i+1)$ is a return, and *loc* otherwise. Note that this word is well-defined because jump-edges represent non-trivial forward leaps.

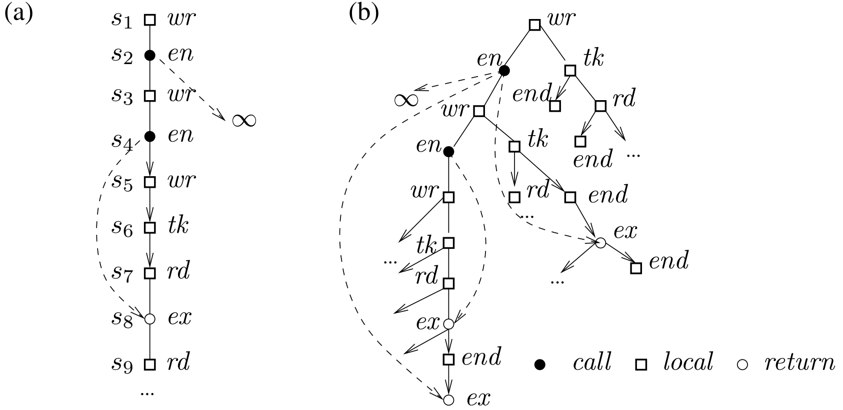



**Figure 1.** (a) A nested word (b) A nested tree

While modeling a program execution, a tagged word defines the sequence of types (call, return or local) of actions in this execution. We note that the construction of $Tag(\mathcal{W})$ requires us to know the jump-edge relation $\hookrightarrow$. More interestingly, the jump-edges in $\mathcal{W}$ are completely captured by the tagged word $(w, \eta)$ of $\mathcal{W}$, so that we can *reconstruct* a nested word from its tagged word. To see why, call a word $\beta \in I^*$ *balanced* if it is of the form $\beta := \beta\beta \mid call.\beta.ret \mid loc$, and define a relation $\hookrightarrow' \subseteq \mathbb{N} \times \mathbb{N}$ as: for all $i < j - 1$, $i \hookrightarrow' j$ iff $i$ is the greatest integer such that the word $\eta(i, i+1).\eta(i+1, i+2) \ldots \eta(j-1, j)$ is balanced. It is easily verified that $\hookrightarrow' = \hookrightarrow$.

Let us denote the set of (finite and infinite) nested words over $\Sigma$ as $NW(\Sigma)$. A language of nested words over $\Sigma$ is a subset of $NW(\Sigma)$.

### 2.2. Nested trees

While nested words are suitable for linear-time reasoning, *nested trees* are necessary to specify branching requirements. Such a structure is obtained by adding jump-edges to an infinite tree whose paths encode all possible executions of the program. As for nested words, jump-edges in nested trees do not cross, and calls and returns are defined respectively as sources and targets of jump-edges. In addition, since a procedure call may not return along all possible program paths, a call-node $s$ may have jump-successors along some, but not all, paths from it. If this is the case, we add a jump-edge from $s$ to a special node $\infty$.

Formally, let $T = (S, r, \rightarrow)$ be an unordered infinite tree with node set $S$, root $r$ and edge relation $\rightarrow \subseteq S \times S$. Let $\xrightarrow{+}$ denote the transitive (but not reflexive) closure of the edge relation, and let a (finite or infinite) *path* in $T$ from node $s_1$ be a (finite or infinite) sequence $\pi = s_1 s_2 \ldots s_n \ldots$ over $S$, where $n \geq 2$ and $s_i \rightarrow s_{i+1}$ for all $1 \leq i$.

A *nested tree* is a directed acyclic graph $(T, \hookrightarrow)$, where $\hookrightarrow \subseteq T \times (T \cup \infty)$ is a set of jump-edges. A node $s$ such that $s \hookrightarrow t$ or $s \hookrightarrow \infty$ (similarly $t \hookrightarrow s$) for some $t$ is a *call* (*return*) node; the remaining nodes are said to be *local*. The intuition is that if $s \hookrightarrow t$, then a call at $s$ returns at $t$; if $s \hookrightarrow \infty$, then there exists a path from $s$ along which the call at $s$ never returns. We note that the sets of call, return and local nodes are disjoint. The jump-edges must satisfy:

1. if $s \hookrightarrow t$, then $s \xrightarrow{+} t$, and we do not have $s \rightarrow t$ (in other words, jump-edges represent non-trivial forward jumps);
2. if $s \hookrightarrow t$ and $s \hookrightarrow t'$, then neither $t \xrightarrow{+} t'$ nor $t' \xrightarrow{+} t$ (this captures the intuition that a call-node has at most one matching return along every path from it);
3. if $s \hookrightarrow t$ and $s' \hookrightarrow t$, then $s = s'$ (every return node has a unique matching call);
4. for every call node $s$, one of the following holds: (a) on every path from $s$, there is a node $t$ such that $s \hookrightarrow t$, and (b) $s \hookrightarrow \infty$ (a call either returns along all paths, or does not);
5. if there is a path $\pi$ such that for nodes $s, t, s', t'$ lying on $\pi$ we have $s \xrightarrow{+} s'$, $s \hookrightarrow t$, and $s' \hookrightarrow t'$, then either $t \xrightarrow{+} s'$ or $t' \xrightarrow{+} t$ (jump-edges along a path do not cross);
6. for every pair of call-nodes $s, s'$ on a path $\pi$ such that $s \xrightarrow{+} s'$, if there is no node $t$ on $\pi$ such that $s' \hookrightarrow t$, then a node $t'$ on $\pi$ can satisfy $s \hookrightarrow t'$ only if $t' \xrightarrow{+} s'$ (if a call does not return, neither do the calls pending when it was invoked).

For an alphabet $\Sigma$, a $\Sigma$-labeled nested tree is a structure $\mathcal{T} = (T, \hookrightarrow, \lambda)$, where $(T, \hookrightarrow)$ is a nested tree with node set $S$, and $\lambda : S \rightarrow \Sigma$ is a node-labeling function. All nested trees in this paper are $\Sigma$-labeled.

Fig. 1-(b) shows a part of the tree unfolding of our example. Note that some of the maximal paths are finite—these capture terminating executions of the program—and some are not. Note in particular how a call may return along some paths from it, and yet not on some others. A path in the nested tree that takes a jump-edge whenever possible is interpreted as a local path through a procedure.

If $s \hookrightarrow t$, then $s$ is the jump-predecessor of $t$ and $t$ the jump-successor of $s$. Edges from a call node and to a return node are known as *call* and *return* edges; the remaining edges are *local*. The fact that an edge $(s, t)$ exists and is a call, return or local edge is denoted by $s \xrightarrow{call} t$, $s \xrightarrow{ret} t$, or $s \xrightarrow{loc} t$. For a nested tree $\mathcal{T} = (T, \hookrightarrow, \lambda)$ with edge set $E$, the *tagged tree* of $\mathcal{T}$ is the node and edge-labeled tree $Tag(\mathcal{T}) = (T, \lambda, \eta : E \rightarrow \{call, ret, loc\})$, where $\eta(s, t) = a$ iff $s \xrightarrow{a} t$.

A few observations: first, the sets of call, return and local edges define a partition of the set of tree edges. Second, if $s \xrightarrow{ret} s_1$ and $s \xrightarrow{ret} s_2$ for distinct $s_1$ and $s_2$, then $s_1$ and $s_2$ have the same jump-predecessor. Third, the jump-edges in a nested tree are completely captured by the edge labeling in the corresponding structured tree, so that we can reconstruct a nested tree $\mathcal{T}$ from $Tag(\mathcal{T})$.

Let $NT(\Sigma)$ be the set of $\Sigma$-labeled nested trees. A language of nested trees is a subset of $NT(\Sigma)$.

### 2.3. *Nested state machines*

Now we define our program abstractions: *nested state machines* (NSMs). Like push-down system and recursive state machines [1], NSMs are suitable for precisely modeling changes to the program stack due to procedure calls and returns. The main difference is that the semantics of an NSM is defined using nested structures rather than a stack and a configuration graph.

Let $AP$ be a fixed set of atomic propositions, and let us set $\Sigma = 2^{AP}$ as an alphabet of *observables*. A *nested state machine* (NSM) is a tuple $\mathcal{M} = \langle V, v_{in}, \kappa, \Delta_{loc}, \Delta_{call}, \Delta_{ret} \rangle$,

where $V$ is a finite set of states, $v_{in} \in V$ is the *initial state*, the map $\kappa : V \to \Sigma$ labels each state with what is observable at it, and $\Delta_{loc} \subseteq V \times V$, $\Delta_{call} \subseteq V \times V$, and $\Delta_{ret} \subseteq V \times V \times V$ are respectively the *local*, *call*, and *return* transition relations.

A transition is said to be *from state* $v$ if it is of the form $(v, v')$ or $(v, v', v'')$, for some $v', v'' \in V$. If $(v, v') \in \Delta_{loc}$ for some $v, v' \in V$, then we write $v \xrightarrow{loc} v'$; if $(v, v') \in \Delta_{call}$, we write $v \xrightarrow{call} v'$; if $(v, v', v'') \in \Delta_{ret}$, we write $(v, v') \xrightarrow{ret} v''$. Intuitively, while modeling a program by an NSM, a transition $(v, v')$ in $\Delta_{call}$ models a procedure call that pushes the current state on the stack, and a transition $(v, v')$ in $\Delta_{loc}$ models a local action (a move that does not modify the stack). In a return transition $(v, v', v'')$, the states $v$ and $v''$ are respectively the current and target states, and $v'$ is the state from which the last "unmatched" call-move was made. The intuition is that $v'$ is on top of the stack right before the return-move, which pops it off the stack.

Let us now abstract our example program into a nested state machine $\mathcal{M}_{foo}$. The abstraction simply captures control flow in the program, and consequently, has states $v_1, v_2, v_3, v_4$, and $v_5$ corresponding to lines L1, L2, L3, L4, and L5. We also have a state $v_2'$ to which control returns after the call at L2 is completed. Now, let us have propositions $rd$, $wr$, $tk$, $en$, $ex$, and $end$ that hold respectively iff the current state represents a read, write, think statement, procedure call, return point after a call, and return instruction. More precisely, $\kappa(v_1) = \{wr\}$, $\kappa(v_2) = \{en\}$, $\kappa(v_2') = \{ex\}$, $\kappa(v_3) = \{tk\}$, $\kappa(v_4) = \{rd\}$, and $\kappa(v_5) = \{end\}$ (for easier reading, we will, from now on, abbreviate singletons such as $\{rd\}$ just as $rd$).

The transition relations of $\mathcal{M}_{foo}$ are given by:

- $\Delta_{call} = \{(v_2, v_1)\}$
- $\Delta_{loc} = \{(v_1, v_2), (v_1, v_3), (v_2', v_4), (v_2', v_5), (v_3, v_4), (v_3, v_5), (v_4, v_4), (v_4, v_5)\}$, and
- $\Delta_{ret} = \{(v_5, v_2, v_2')\}$.

**Linear-time semantics** The linear-time semantics of a nested state machine $\mathcal{M} = \langle V, v_{in}, \kappa, \Delta_{loc}, \Delta_{call}, \Delta_{ret} \rangle$ is given by a language $\mathcal{L}(\mathcal{M})$ of *traces*; this is a language of nested words over the alphabet $2^{AP}$. First consider the language $\mathcal{L}^V(\mathcal{M})$ of *nested executions* of $\mathcal{M}$, comprising nested words over the alphabet $V$ of states. A nested word $\mathcal{W} = (w, \hookrightarrow)$ is in $\mathcal{L}^V(\mathcal{M})$ iff the tagged word $(w, \eta)$ of $\mathcal{W}$ is such that $w(0) = v_{in}$, and for all $i \geq 0$, (1) if $\eta(i, i+1) \in \{call, loc\}$, then $w(i) \xrightarrow{\eta(i,i+1)} w(i+1)$; and (2) if $\eta(i, i+1) = ret$, then there is a $j$ such that $j \hookrightarrow (i+1)$ and we have $(w(i), w(j)) \xrightarrow{ret} w(i+1)$. Now, a trace produced by an execution is the sequence of observables it corresponds to. Accordingly, the trace language $\mathcal{L}(\mathcal{M})$ of $\mathcal{M}$ is defined as $\{(w', \hookrightarrow) :$ for some $(w, \hookrightarrow) \in \mathcal{L}^V(\mathcal{M})$ and all $i \geq 0, w'(i) = \kappa(w_i)\}$. For example, the nested word in Fig. 1-(a) belongs to the trace language of $\mathcal{M}_{foo}$.

**Branching-time semantics** The branching-time semantics of $\mathcal{M}$ is defined via a $2^{AP}$-labeled tree $\mathcal{T}(\mathcal{M})$, known as the *unfolding* of $\mathcal{M}$. For branching-time semantics to be well-defined, an NSM must satisfy an additional condition: every transition from a state $v$ is of the same type (call, return, or local). The idea is to not allow the same node to be a call along one path and, say, a return along another. Note that this is the case in NSMs whose states model statements in programs.

Now consider the $V$-labeled nested tree $\mathcal{T}^V(\mathcal{M}) = (T, \hookrightarrow, \lambda)$, known as the *execution tree*, that is the unique nested tree satisfying the following conditions:

1. if $r$ is the root of $T$, then $\lambda(r) = v_{in}$;
2. every node $s$ has precisely one child $t$ for every distinct transition in $\mathcal{M}$ from $\lambda(s)$;
3. for every pair of nodes $s$ and $t$, if $s \xrightarrow{a} t$, for $a \in \{call, loc\}$, in the tagged tree of this nested tree, then we have $\lambda(s) \xrightarrow{a} \lambda(t)$ in $\mathcal{M}$;
4. for every $s, t$, if $s \xrightarrow{ret} t$ in the tagged tree, then there is a node $t'$ such that $t' \hookrightarrow t$ and $(\lambda(s), \lambda(t')) \xrightarrow{ret} \lambda(t)$ in $\mathcal{M}$.

Note that this definition is possible as we assume transitions from the same state of $\mathcal{M}$ to be of the same type. Now we have $\mathcal{T}(\mathcal{M}) = (T, \hookrightarrow, \lambda')$, where $\lambda'(s) = \kappa(\lambda(s))$ for all nodes $s$. For example, the nested tree in Fig. 1-(b) is the unfolding of $\mathcal{M}_{foo}$.

## 3. Specifications

In this section, we define automata and a fixpoint logic on nested words and trees, and explore their applications to program specification. Automata on nested words are useful for linear-time model-checking, where the question is: "is the language of nested traces of the abstraction (an NSM) included in the language of nested words allowed by the specification?" In the branching-time case, model checking question is: "is the unfolding of the NSM a member of the set of nested trees allowed by the specification?" Our hypothesis is these altered views of the model-checking problem are better suited to software verification.

### *3.1. Automata on nested words*

We start with finite automata on nested words [7,6]. A *nested Büchi word automaton* (NWA) over an alphabet $\Sigma$ is a tuple $\mathcal{A} = \langle Q, \Sigma, q_{in}, \delta_{loc}, \delta_{call}, \delta_{ret}, G\rangle$, where $Q$ is a set of states $Q$, $q_{in}$ is the initial state, and $\delta_{loc} \subseteq Q \times \Sigma \times Q$, $\delta_{call} \subseteq Q \times \Sigma \times Q$, and $\delta_{ret} \subseteq Q \times Q \times \Sigma \times Q$ are the local, call and return transition relations. The *Büchi acceptance condition* $G \subseteq Q$ is a set of *accepting states*. If $(q, \sigma, q') \in \delta_{loc}$ for some $q, q' \in Q$ and $\sigma \in \Sigma$, then we write $q \xrightarrow{loc,\sigma} q'$; if $(q, \sigma, q') \in \delta_{call}$, we write $q \xrightarrow{call,\sigma} q'$; if $(q, q', \sigma, q'') \in \delta_{ret}$, we write $(q, q') \xrightarrow{ret,\sigma} q''$.

The automaton $\mathcal{A}$ starts in the initial state, and reads a nested word from left to right. At a call or local position, the current state is determined by the state and the input symbol (in case of traces of NSMs, the observable) at the previous position, while at a return position, the current state can additionally depend on the state of the run just before processing the symbol at the jump-predecessor. Formally, a *run* $\rho$ of the automaton $\mathcal{A}$ over a nested word $\mathcal{W} = (\sigma_1\sigma_2 \ldots, \hookrightarrow)$ is an infinite sequence $q_0, q_1, q_2, \ldots$ over $Q$ such that $q_0 = q_{in}$, and:

- for all $i \geq 0$, if $i$ is a call position of $\mathcal{W}$, then $(q_i, \sigma_i, q_{i+1}) \in \delta_{call}$;
- for all $i \geq 0$, if $i$ is a local position, then $(q_i, \sigma_i, q_{i+1}) \in \delta_{loc}$;
- for $i \geq 2$, if $i$ is a return position with jump-predecessor $j$, then $(q_{i-1}, q_{j-1}, \sigma_i, q_i) \in \delta_{ret}$.

The automaton $\mathcal{A}$ accepts a finite nested word $\mathcal{W}$ if it has a run $q_0, q_1, q_2, \ldots q_n$ over $\mathcal{W}$ such that $q_n \in G$. An infinite nested word is accepted if there is a run $q_0, q_1, q_2, \ldots$

where a state $q \in G$ is visited infinitely often. The language $\mathcal{L}(\mathcal{A})$ of a nested-word automaton $\mathcal{A}$ is the set of nested words it accepts.

A language $L$ of nested words over $\Sigma$ is *regular* if there exists a nested-word automaton $\mathcal{A}$ over $\Sigma$ such that $L = \mathcal{L}(\mathcal{A})$. Observe that if $L$ is a regular language of words over $\Sigma$, then $\{(w, \hookrightarrow) \mid w \in L\}$ is a regular language of nested words. Conversely, if $L$ is a regular language of nested words, then $\{w \mid (w, \hookrightarrow) \in L \text{ for some } \hookrightarrow\}$ is a context-free language of words, but need not be regular.

Let us now see an example of how NWAs may be used for specification. Consider the following property to be tested on our running example: "in every execution of the program, every occurrence of *write(x)* is followed (not necessarily immediately) by an occurrence of *read(x)*." This property can be expressed by a finite-state, Büchi *word automaton*. As before, we have $\Sigma = \{wr, rd, en, ex, tk, end\}$. The automaton $\mathcal{S}$ has states $q_1$ and $q_2$; the initial state is $q_1$. The automaton has transitions $q_1 \xrightarrow{wr} q_2$, $q_2 \xrightarrow{wr} q_2$, $q_2 \xrightarrow{rd} q_1$, and $q_1 \xrightarrow{rd} q_1$ (on all input symbols other that $wr$ and $rd$, $\mathcal{S}$ stays at the state from which the transition fires). The idea is that at the state $q_2$, $\mathcal{S}$ expects to see a read some time in the future. Now, we have a single Büchi accepting state $q_1$, which means the automaton cannot get stuck in state $q_2$, thus accepting precisely the set of traces satisfying our requirement.

However, consider the property: "in every execution of the program, every occurrence of *write(x)* is followed (not necessarily immediately) by an occurrence of *read(x) in the same procedural context* (i.e., before control returns from the current context, and not within the scope of new contexts transitively spawned from this context via calls)." A finite-state word automaton cannot state this requirement, not being able to reason about the balanced-parentheses structure of calls and returns. On the other hand, this property can be expressed simply by a NWA $\mathcal{A}$ with states $q_1$, $q_2$ and $q_e$—here, $q_2$ is the state where $\mathcal{A}$ expects to see a read action in the same context at some point in the future, $q_e$ is an error state, and $q_1$ is the state where there is no requirement for the current context. The initial state is $q_1$. As for transitions:

- we have $q_1 \xrightarrow{loc, wr} q_2$, $q_1 \xrightarrow{loc, rd} q_1$, $q_2 \xrightarrow{loc, rd} q_1$, and $q_2 \xrightarrow{loc, wr} q_2$ (these transitions are for the same reason as in $\mathcal{S}$);
- we have $q_1 \xrightarrow{call, en} q_1$ and $q_2 \xrightarrow{call, en} q_1$ (as the requirement only relates reads and writes in the same context, we need to "reset" the state when a new context starts due to a call);
- for $q' \in \{q_1, q_2\}$, we have $(q_1, q') \xrightarrow{ret, end} q'$ (suppose we have reached the end of a context. So long as there is no requirement pending within this context, we must, on return, restore the state to where it was before the call. Of course, this transition is only fired in contexts that are not at the top-level.) We also have, for $q' \in \{q_1, q_2\}$, $(q_2, q') \xrightarrow{ret, end} q_e$ (in other words, it is an error to end a context before fulfilling a pending requirement).
- Also, for $q' \in \{q_1, q_2\}$, we have $q' \xrightarrow{loc, tk} q'$ and $q' \xrightarrow{loc, ex} q'$.

The single Büchi accepting state, as before, is $q_1$.

More "realistic" requirements that may be stated using automata on nested words include:

- *Pre/post-conditions:* Consider partial and total correctness requirements based on pre/post-conditions, which show up in Hoare-Floyd-style program verification as

well as in modern interface specification languages such JML [11] and SAL [14]. Partial correctness for a procedure A asserts that if precondition *Pre* is satisfied when A is called, then if A terminates, postcondition *Post* holds upon return. Total correctness, additionally, requires A to terminate. If program executions are modeled using nested words, these properties are just assertions involving the current state and jump-successors, and can be easily stated using automata on nested words.
- *Access control:* Specifications such as "in all executions of a proggram, a procedure A can access a database only if all the frames on the stack have high privilege" are useful in software security and are partially enforced at runtime in programming languages such as Java. Such "stack inspection" properties cannot be stated using traditional temporal logics and automata on words. It can be shown, however, that they are easily stated using nested word languages.
- *Boundedness:* Using nested word languages, we can state requirements such as "the height of the stack is bounded by $k$ along all executions," useful to ensure that there is no stack overflow. Another requirement of this type: "every call in every program execution eventually returns."

We will now list a few properties of regular languages of nested words. The details may be found in the original papers [7,6,5].

- The class of regular languages of nested words is (effectively) closed under union, intersection, complementation, and projection.
- Language membership, inclusion, and emptiness are decidable.
- Automata on finite nested words can be determinized.
- Automata on finite nested words can be characterized using Myhill-Nerode-style congruences, and a subclass of these may be reduced to a unique minimum form.
- Automata on finite or infinite nested words are expressively equivalent to monadic second order logic (MSO) augmented with a binary "jump" predicate capturing the jump-edge relation in nested words. This generalizes the equivalence of regular word languages and the logic S1S.

An alternative way to specify linear-time behaviors of nested executions of programs is to use *temporal logics on nested words*. First consider the logic CARET [4], which may be viewed as an extension of LTL on nested words. Like LTL, this logic has formulas such as $\bigcirc\varphi$ (the property $\varphi$ holds at the next time point), $\Box\varphi$ ($\varphi$ holds at every point in the present and future), and $\Diamond\varphi$ ($\varphi$ holds eventually). The formulas are evaluated as LTL formulas on the word $w$ in a nested word $\mathcal{W} = (w, \hookrightarrow)$. In addition, CARET defines the notion of an "abstract successor" in a nested word—the abstract successor of a call position is its jump-successor, and that of a return or local position is its successor—and has formulas such as $\bigcirc^a\varphi$ (the property $\varphi$ holds at the abstract successor) and $\Diamond^a\varphi$ ($\varphi$ holds at some future point in the current context). The full syntax and semantics may be found in the original reference. For a concrete example, consider the property we specified earlier using nested word automata. In CARET, this specification is given by a formula $\varphi = \Box(wr \Rightarrow \Diamond^a rd)$, which is interpreted as "every write is followed (not necessarily immediately) by a read in the same context," and asserted at the initial program state. So far as model-checking goes, every CARET specification may be compiled into an equivalent (and, at worst, exponentially larger) Büchi NWA, so that the model-checking problem for CARET reduces to that for NWAs.

More recently, the linear-time $\mu$-calculus has been extended to nested word models [13]. This logic has modalities $\bigcirc$ and $\bigcirc^a$, which assert requirements respectively at the successor and abstract successor of a position, and, in addition, has set-valued variables $x$ and fixpoint formulas such as $\mu X.\varphi(X)$. We will not go into the details in this paper, but recall that a property "a position satisfying $rd$ is reached eventually" can be stated in the linear-time $\mu$-calculus as $\varphi = \mu X.(rd \vee \bigcirc X)$ (the notation is standard and hence not defined in detail). A property "$rd$ is reached eventually in the current context" is expressed in the linear-time $\mu$-calculus on nested words by the formula $\varphi = \mu X.(rd \vee \bigcirc^a X)$. It turns out that this logic has a number of attractive properties—for example, it is expressively equivalent to MSO-logic interpreted on nested words, and collapses to its alternation-free fragment on finite nested words. Like CARET, formulas in this logic can also be compiled into equivalent NWAs.

### 3.2. A fixpoint calculus on nested trees

Now we introduce a fixpoint calculus, known as NT-$\mu$, for nested trees [2]. This logic may be viewed as an analog of the modal $\mu$-calculus for nested trees. Recall that a $\mu$-calculus formula is interpreted at a state $s$ of a program, or, equivalently, on the full subtree rooted at a node corresponding to $s$ in the program's tree unfolding. NT-$\mu$ is interpreted on substructures of nested trees wholly contained within "procedural" contexts; such a structure models the branching behavior of a program from a state $s$ to each exit point of its context. Also, to demand different temporal requirements at different exits, we introduce a *coloring* of these exits—intuitively, an exit gets color $i$ if it is to satisfy the $i$-th requirement.

Formally, let a node $t$ of $\mathcal{T}$ be called a *matching exit* of a node $s$ if there is an $s'$ such that $s' \xrightarrow{+} s$ and $s' \hookrightarrow t$, and there are no $s'', t''$ such that $s' \xrightarrow{+} s'' \xrightarrow{+} s \xrightarrow{+} t''$, and $s'' \hookrightarrow t''$. Intuitively, a matching exit of $s$ is the first "unmatched" return along some path from $s$—for instance, in Fig. 1-(a), the node $s_8$ is the single matching exit of the nodes $s_5$, $s_6$, and $s_7$. Let the set of matching exits of $s$ be denoted by $ME(s)$. For a non-negative integer $k$, a *summary* s in $\mathcal{T}$ is a tuple $\langle s, U_1, U_2, \ldots, U_k \rangle$, where $s$ is a node, $k \geq 0$, and $U_1, U_2, \ldots, U_k \subseteq ME(s)$ (such a summary is said to be *rooted* at $s$). The set of summaries in a nested tree $\mathcal{T}$ is denoted by $Summ^{\mathcal{T}}$. Note that such colored summaries are defined for all $s$, not just "entry" nodes of procedures.

In addition to being interpreted over summaries, the logic NT-$\mu$, can distinguish between call, return and local edges in a nested tree via modalities such as $\langle call \rangle$, $\langle ret \rangle$, and $\langle loc \rangle$. Also, an NT-$\mu$ formula can enforce different "return conditions" at differently colored returns by passing subformulas as "parameters" to $call$ modalities. Let $AP$ be a finite set of atomic propositions, $Var$ be a finite set of *variables*, and $R_1, R_2, \ldots$ be a countable, ordered set of *markers*. For $p \in AP$, $X \in Var$, and $m \geq 0$, formulas $\varphi$ of NT-$\mu$ are defined by:

$$\varphi, \psi_i := p \mid \neg p \mid X \mid \langle ret \rangle(R_i) \mid [ret](R_i) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu X.\varphi \mid \nu X.\varphi \mid$$
$$\langle call \rangle(\varphi)\{\psi_1, \psi_2, ..., \psi_m\} \mid [call](\varphi)\{\psi_1, \psi_2, ..., \psi_m\} \mid \langle loc \rangle\, \varphi \mid [loc]\, \varphi.$$

Intuitively, the markers $R_i$ in a formula are bound by $\langle call \rangle$ and $[call]$ modalities, and variables $X$ are bound by fixpoint quantifiers $\mu X$ and $\nu X$. The set of free variables is defined in the usual way. Also, we require our $call$-formulas to bind all the markers in their scope—for example, formulas such as $\varphi = \langle call \rangle(p \wedge \langle ret \rangle R_1)\{q\} \wedge \langle ret \rangle R_1$

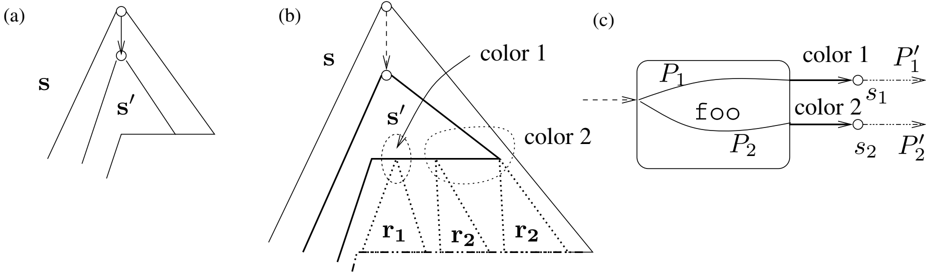



**Figure 2.** (a) Local modalities (b) Call modalities (c) Matching contexts.

are not permitted. A formula that satisfies this criterion is called *closed* if it has no free variables. The *arity* of a formula $\varphi$ is the maximum $m$ such that $\varphi$ has a subformula $\langle call \rangle \varphi' \{\psi_1, \ldots, \psi_m\}$ or $[call]\varphi'\{\psi_1, \ldots, \psi_m\}$. Also, we define the constants $tt$ and $ff$ in the standard way.

Like in the $\mu$-calculus, formulas in NT-$\mu$ encode sets, in this case sets of summaries. Also like in the $\mu$-calculus, modalities and boolean and fixed-point operators allow us to encode computations on these sets.

To understand the semantics of local (e.g. $\langle loc \rangle$) modalities in NT-$\mu$, consider a node $s$ in a nested tree with a local edge to a node $s'$. Note that $ME(s') \subseteq ME(s)$, and consider two summaries $\mathsf{s}$ and $\mathsf{s}'$ rooted respectively at $s$ and $s'$. Now look at Fig. 2-a. Note that the substructure $\mathcal{T}_{s'}$ captured by the summary $\mathsf{s}'$ "hangs"' from the substructure for $\mathsf{s}$ by a local edge; additionally, (1) every leaf of $\mathcal{T}_{s'}$ is a leaf of $\mathcal{T}_s$, and (2) such a leaf gets the same color in $\mathsf{s}$ and $\mathsf{s}'$. A formula $\langle loc \rangle \varphi$ asserted at $\mathsf{s}$ requires some $\mathsf{s}'$ as above to satisfy $\varphi$.

Succession along call edges is more complex, because along such an edge, a new context gets defined. Suppose we have $s \xrightarrow{call} s'$, and let there be no other edges from $s$. Consider the summary $\mathsf{s} = \langle s, \{s_1\}, \{s_2, s_3\} \rangle$, and suppose we want to assert a 2-parameter call formula $\langle call \rangle \varphi' \{p_1, p_2\}$ at $s$. This requires us to consider a 2-colored summary of the context starting at $s'$, where matching returns of $s'$ satisfying $p_1$ and $p_2$ are respectively marked by colors 1 and 2. Our formula requires that $\mathsf{s}'$ satisfies $\varphi'$. In general, we could have formulas of the form $\varphi = \langle call \rangle \varphi' \{\psi_1, \psi_2, \ldots, \psi_k\}$, where $\psi_i$ are arbitrary NT-$\mu$ formulas. We find that the above requires a split of the nested tree $\mathcal{T}_s$ for summary $\mathsf{s}$ in the way shown in Fig. 2-b. The root of this tree must have a $call$-edge to the root of the tree for $\mathsf{s}'$, which must satisfy $\varphi$. At each leaf of $\mathcal{T}_{s'}$ colored $i$, we must be able to *concatenate* a summary tree $\mathcal{T}_{s''}$ satisfying $\psi_i$ such that (1) every leaf in $\mathcal{T}_{s''}$ is a leaf of $\mathcal{T}_s$, and (2) each such leaf gets the same set of colors in $\mathcal{T}_s$ and $\mathcal{T}_{s''}$.

The return modalities are used to assert that we return at a point colored $i$. As the binding of these colors to requirements gets fixed at a context calling the current context, the $ret$-modalities let us relate a path in the latter with the continuation of a path in the former. For instance, in Fig. 2-c, where the rectangle abstracts the part of a program unfolding within the body of a procedure `foo`, the marking of return points $s_1$ and $s_2$ by colors 1 and 2 is visible inside `foo` as well as at the call site of `foo`. This lets us match paths $P_1$ and $P_2$ inside `foo` respectively with paths $P_1'$ and $P_2'$ in the calling procedure. This lets NT-$\mu$ capture the pushdown structure of branching-time runs of a procedural program.

Let us now describe the semantics of NT-$\mu$ formally. An NT-$\mu$ formula $\varphi$ is interpreted in an *environment* that interprets variables in $Free(\varphi)$ as sets of summaries in a nested tree $\mathcal{T}$. Formally, an *environment* is a map $\mathcal{E} : Free(\varphi) \rightarrow 2^{Summ^{\mathcal{T}}}$. Let us write $[\![\varphi]\!]_{\mathcal{E}}^{\mathcal{T}}$ to denote the set of summaries in $\mathcal{T}$ satisfying $\varphi$ in environment $\mathcal{E}$ (usually $\mathcal{T}$ will be understood from the context, and we will simply write $[\![\varphi]\!]_{\mathcal{E}}$). For a summary $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle$, where $s \in S$ and $U_i \subseteq ME(s)$ for all $i$, $\mathbf{s}$ satisfies $\varphi$, i.e., $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$, iff one of the following holds:

- $\varphi = p \in AP$ and $p \in \lambda(s)$
- $\varphi = \neg p$ for some $p \in AP$, and $p \notin \lambda(s)$
- $\varphi = X$, and $\mathbf{s} \in \mathcal{E}(X)$
- $\varphi = \varphi_1 \vee \varphi_2$ such that $\mathbf{s} \in [\![\varphi_1]\!]_{\mathcal{E}}$ or $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
- $\varphi = \varphi_1 \wedge \varphi_2$ such that $\mathbf{s} \in [\![\varphi_1]\!]_{\mathcal{E}}$ and $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
- $\varphi = \langle call\rangle\varphi'\{\psi_1, \psi_2, ..., \psi_m\}$, and there is a $t \in S$ such that (1) $s \xrightarrow{call} t$, and (2) the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_m\rangle$, where for all $1 \leq i \leq m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s')\rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = [call]\ \varphi'\{\psi_1, \psi_2, ..., \psi_m\}$, and for all $t \in S$ such that $s \xrightarrow{call} t$, the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_m\rangle$, where for all $1 \leq i \leq m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s')\rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = \langle loc\rangle\ \varphi'$, and there is a $t \in S$ such that $s \xrightarrow{loc} t$ and the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_k\rangle$, where $V_i = ME(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = [loc]\ \varphi'$, and for all $t \in S$ such that $s \xrightarrow{loc} t$, the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_k\rangle$, where $V_i = ME(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = \langle ret\rangle\ R_i$, and there is a $t \in S$ such that $s \xrightarrow{ret} t$ and $t \in U_i$
- $\varphi = [ret]\ R_i$, and for all $t \in S$ such that $s \xrightarrow{ret} t$, we have $t \in U_i$
- $\varphi = \mu X.\varphi'$, and $\mathbf{s} \in \mathbf{S}$ for all $\mathbf{S} \subseteq Summ^{\mathcal{T}}$ satisfying $[\![\varphi']\!]_{\mathcal{E}[X:=\mathbf{S}]} \subseteq \mathbf{S}$
- $\varphi = \nu X.\varphi'$, and there is some $\mathbf{S} \subseteq Summ^{\mathcal{T}}$ such that (1) $\mathbf{S} \subseteq [\![\varphi']\!]_{\mathcal{E}[X:=\mathbf{S}]}$ and (2) $\mathbf{s} \in \mathbf{S}$.

Here $\mathcal{E}[X := \mathbf{S}]$ is the environment $\mathcal{E}'$ such that (1) $\mathcal{E}'(X) = \mathbf{S}$, and (2) $\mathcal{E}'(Y) = \mathcal{E}(Y)$ for all variables $Y \neq X$. We say a node $s$ *satisfies* a formula $\varphi$ if the 0-colored summary $\langle s\rangle$ satisfies $\varphi$. A nested tree $\mathcal{T}$ rooted at $s_0$ is said satisfy $\varphi$ if $s_0$ satisfies $\varphi$ (we denote this by $\mathcal{T} \models \varphi$). The language of $\varphi$, denoted by $\mathcal{L}(\varphi)$, is the set of nested trees satisfying $\varphi$.

While formulas such as $\neg\varphi$ (negation of $\varphi$) are not directly given by the syntax of NT-$\mu$, we can show that *closed* formulas of NT-$\mu$ are closed under negation. Also, note that the semantics of closed NT-$\mu$ formulas is independent of the environment. Also, the semantics of such a formula $\varphi$ does not depend on current color assignments; in other words, a summary $\mathbf{s} = \langle s, U_1, \ldots, U_k\rangle$ satisfies a closed formula iff $\langle s\rangle$ satisfies $\varphi$. Consequently, when $\varphi$ is closed, we can infer that "node $s$ satisfies $\varphi$" from "summary $\mathbf{s}$ satisfies $\varphi$." Finally, every NT-$\mu$ formula $\varphi(X)$ with a free variable $X$ can be viewed as a map $\varphi(X) : 2^{Summ^{\mathcal{T}}} \rightarrow 2^{Summ^{\mathcal{T}}}$ defined as follows: for all environments $\mathcal{E}$ and all summary sets $\mathbf{S} \subseteq Summ^{\mathcal{T}}$, $\varphi(X)(\mathbf{S}) = [\![\varphi(X)]\!]_{\mathcal{E}[X:=\mathbf{S}]}$. It is not hard to verify that this map is monotonic, and that therefore, by the Tarski-Knaster theorem, its least and greatest fixed points exist. The formulas $\mu X.\varphi(X)$ and $\nu X.\varphi(X)$ respectively evaluate to these two sets. This means the set of summaries satisfying $\mu X.\varphi(X)$, for instance, lies in the sequence of summary sets $\emptyset, \varphi(\emptyset), \varphi(\varphi(\emptyset)), \ldots$.

Just as the $\mu$-calculus can encode linear-time logics such as LTL as well as branching-time logics such as CTL, NT-$\mu$ can capture linear and branching properties on nested trees. Let us now specify our example program using a couple of requirements. Consider the simple property $Reach$ asserted at the initial state of the program: "the instruction *read(x)* is reachable from the current node." Let us continue to use the atomic propositions $rd$, $wr$, etc. that we have been using through the paper. This property may be stated in the $\mu$-calculus as $\varphi_{Reach} = (\mu X.rd \vee \langle\rangle X)$ (the notation is standard—for instance, $\langle\rangle\varphi$ holds at a node iff $\varphi$ holds at a node reached by some edge). However, let us try to define it using NT-$\mu$.

First consider a nontrivial witness $\pi$ for $Reach$ that starts with an edge $s \xrightarrow{call} s'$. There are two possibilities: (1) a node satisfying $rd$ is reached in the new context or a context called transitively from it, and (2) a matching return $s''$ of $s'$ is reached, and at $s''$, $Reach$ is once again satisfied.

To deal with case (2), we mark a matching return that leads to $rd$ by color 1. Let $X$ store the set of summaries of form $\langle s''\rangle$, where $s''$ satisfies $Reach$. Then we want the summary $\langle s, ME(s)\rangle$ to satisfy $\langle call\rangle\varphi'\{X\}$, where $\varphi'$ states that $s'$ can reach one of its matching returns of color 1. In case (1), there is no return requirement (we do not need the original call to return), and we simply assert $\langle call\rangle X\{\}$.

Before we get to $\varphi'$, note that the formula $\langle loc\rangle X$ captures the case when $\pi$ starts with a local transition. Combining the two cases, the formula we want is $\varphi_{Reach} = \mu X.(rd \vee \langle loc\rangle X \vee \langle call\rangle X\{\} \vee \langle call\rangle\varphi'\{X\})$.

Now observe that $\varphi'$ also expresses reachability, except (1) its target needs to satisfy $\langle ret\rangle R_1$, and (2) this target needs to lie in the *same procedural context* as $s'$. It is easy to verify that: $\varphi' = \mu Y.(\langle ret\rangle R_1 \vee \langle loc\rangle Y \vee \langle call\rangle Y\{Y\})$.

Let us now suppose we are interested in *local reachability*: "a node satisfying $rd$ is reached in the current context." This property cannot be expressed by finite-state automata on words or trees, and hence cannot be captured by the $\mu$-calculus. However, we note that the property $\varphi'$ is very similar in spirit to this property. While we cannot merely substitute $rd$ for $\langle ret\rangle R_1$ in $\varphi'$ to express local reachability of $rd$, a formula for this property is easily obtained by restricting the formula for reachability: $\varphi_{LocalReach} = \mu X.(rd \vee \langle loc\rangle X \vee \langle call\rangle\varphi'\{X\})$.

Note that the highlight of this approach to specification is the way we split a program unfolding along procedure boundaries, specify these "pieces" modularly, and plug the summary specifications so obtained into their call sites. This "interprocedural" reasoning distinguishes it from logics such as the $\mu$-calculus that would reason only about *global* runs of the program.

Also, there is a significant difference in the way fixpoints are computed in NT-$\mu$ and the $\mu$-calculus. Consider the fixpoint computation for the $\mu$-calculus formula $\mu X.(rd \vee \langle\rangle X)$ that expresses reachability of a node satisfying $rd$. The semantics of this formula is given by a set $S_X$ of nodes which is computed iteratively. At the end of the $i$-th step, $S_X$ comprises nodes that have a path with at most $(i-1)$ transitions to a node satisfying $rd$. Contrast this with the evaluation of the outer fixpoint in the NT-$\mu$ formula $\varphi_{Reach}$. Assume that $\varphi'$ (intuitively, the set of "jumps" from calls to returns") has already been evaluated, and consider the set $\mathbf{S_X}$ of summaries for $\varphi_{Reach}$. At the end of the $i$-th phase, this set contains all $\mathrm{s} = \langle s\rangle$ such that $s$ has a path consisting of $(i-1)$ $call$ and $loc$-transitions to a node satisfying $rd$. However, because of the subformula $\langle call\rangle\varphi'\{X\}$, it also includes all $\mathrm{s}$ where $s$ reaches $rd$ via a path of at most $(i-1)$ local and "jump"

transitions. Note how return edges are considered only as part of summaries plugged into the computation.

More details about specification using NT-$\mu$ may be found in the original reference [2]. Here we list some other requirements expressible in NT-$\mu$:

- Any closed $\mu$-calculus formula, as well as any property expressible in CARET or automata on nested words, may be expressed in NT-$\mu$. Consequently, NT-$\mu$ can express pre/post-conditions on procedures, access control requirements involving the stack, and requirements on the height of the stack, as well as traditional linear and branching-time requirements.
- *Interprocedural dataflow requirements:* It is well-known that many classic dataflow analysis problems, such as determining whether an expression is very busy, can be reduced to the problem of finding the set of program points where a certain $\mu$-calculus property holds [20]. However, the $\mu$-calculus is unable to state that an expression is very busy at a program point if it has local as well as global variables and we are interested in interprocedural paths—the reason is that dataflow involving global variables follows a program execution through procedure calls, while dataflow for local variables "jumps" across procedure calls, and the $\mu$-calculus cannot track them both at the same time. On the other hand, the ability of NT-$\mu$ to assert requirements along jump-edges as well as tree edges lets it express such requirements.

We end this discussion by listing some known mathematical properties of NT-$\mu$.

- Generalizing the notion of bisimulation on trees, we may define bisimulation relations on nested trees [2]. Then two nested trees satisfy the same set of closed NT-$\mu$ formulas iff they are bisimilar.
- The satisfiability problem for NT-$\mu$ is undecidable [2].
- Just as the modal $\mu$-calculus is expressively equivalent to alternating parity tree automata, NT-$\mu$ has an automata-theoretic characterization. Generalizing automata on nested words, we can define automata on nested trees; generalizing further, we can define *alternating parity automata on nested trees*. It turns out that every closed formula of NT-$\mu$ has a polynomial translation to such an automaton accepting the same set of nested trees, and vice versa [3].

## 4. Model-checking

In this section, we show how to model-check specifications on nested structures generated by NSMs. Our chosen specification language in this section is the logic NT-$\mu$—the reason is that it can express linear as well as branching-time temporal specifications, and lends itself to iterative, symbolic model-checking. Appealingly, this algorithm follows directly from the operational semantics of the logic and has the same complexity (EXPTIME) as the best algorithms for model-checking CTL or the alternation-free $\mu$-calculus over similar abstractions.

For a specification given by a (closed) NT-$\mu$ formula $\varphi$ and an NSM $\mathcal{M}$ abstracting a program, the *model-checking problem* is to determine if $\mathcal{T}(\mathcal{M})$ satisfies $\varphi$. It is also useful to define the model-checking problem for NWAs: here, a problem instance comprises an NSM $\mathcal{M}$ abstracting a program, and an NWA $\mathcal{A}_\neg$ accepting the nested words

that model program executions that are *not acceptable*. The model-checking problem in this case is whether any of the possible program traces are "bad", i.e., if $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\mathcal{A}_\neg)$ is non-empty. Of course, instead of phrasing the problem this way, we could have also let the instance consist of an NSM and a specification automaton $\mathcal{A}'$, in which case we would have to check if $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{A}')}$ is non-empty. However, complementation of $\mathcal{A}'$, while possible, is costly, and this approach would not be practical.

Now, intersection of the languages of two NWAs is done by a product construction [6]. The model-checking problem thus boils down to checking the emptiness of a Büchi NWA $\mathcal{A}$. Let us now view $\mathcal{A}$ as an NSM where a state is marked by a proposition $g$ iff it is a Büchi accepting state. An NWA on infinite nested words is then non-empty iff there are infinitely many occurrences of $g$ along some path in the unfolding of $\mathcal{A}$, a requirement can be expressed as a fixpoint formula in the $\mu$-calculus, and hence NT-$\mu$. To determine that an NWA on finite nested words is non-empty, we merely need to ensure that a node satisfying $g$ is reachable in this unfolding—an NT-$\mu$ formula for this property is as in the example in Sec. 3.2.

We will now show how to do NT-$\mu$ model-checking for an NSM $\mathcal{M}$ with vertex set $V$ and an NT-$\mu$ formula $\varphi$. Consider a node $s$ in the nested tree $\mathcal{T}^V(\mathcal{M})$. The set $ME(s)$, as well as the return-formulas that hold at a summary s rooted at $s$, depend on states at call nodes on the path from the root to $s$. However, we observe that the history of call-nodes up to $s$ is relevant to a formula only because they may be consulted by return-nodes in the future, and no formula interpreted at s can probe "beyond" the nodes in $ME(s)$. Thus, so far as satisfaction of a formula goes, we are only interested in the *last* "pending" call-node; in fact, the state of the automaton at this node is all that we need to record about the past.

Let us now try to formalize this intuition. First we define the *unmatched call-ancestor* $Anc(s)$ of a node $s$ in a nested tree $\mathcal{T}$. Consider the tagged tree of $\mathcal{T}$, and recall the definition of a balanced word over tags (given in Sec. 2.1). If $t = Anc(s)$, then we require that $t \xrightarrow{call} t'$ for some node $t'$ such that in the tagged tree of $\mathcal{T}$, there is a path from $t'$ to $s$ the edge labels along which concatenate to form a balanced word. Note that every node in a nested tree has at most one unmatched call-ancestor. If a node $s$ does not have such an ancestor, we set $Anc(s) = \perp$.

Now consider two $k$-colored summaries $\mathrm{s} = \langle s, U_1, U_2, \ldots, U_k \rangle$ and $\mathrm{s}' = \langle s', U'_1, U'_2, \ldots, U'_k \rangle$ in the unfolding $\mathcal{T}^V(\mathcal{M}) = (T, \hookrightarrow, \lambda)$ of the NSM $\mathcal{M}$, and let $Anc(s) = t$ and $Anc(s') = t'$, where $t, t'$ can be nodes or the symbol $\perp$ (note that if we have $Anc(s) = \perp$, then $ME(s) = \emptyset$, so that $U_i = \emptyset$ for all $i$).

Now we say s and s$'$ are *NSM-equivalent* (written as $\mathrm{s} \equiv \mathrm{s}'$) if:

- $\lambda(s) = \lambda(s')$;
- either $t = t' = \perp$, or $\lambda(t) = \lambda(t')$;
- for each $1 \leq i \leq k$, there is a bijection $\Omega_i : U_i \to U'_i$ such that for all $u \in U_i$, we have $\lambda(u) = \lambda(\Omega_i(u))$.

It is easily seen that the relation $\equiv$ is an equivalence. We can also prove that any two NSM-equivalent summaries s and s$'$ satisfy the same set of closed NT-$\mu$ formulas.

Now note that the number of equivalence classes that $\equiv$ induces on the set of summaries is bounded! Each such equivalence class may be represented by a tuple $\langle v, v', V_1, \ldots, V_k \rangle$, where $v \in V$, $v' \in V \cup \{\perp\}$, and $V_i \subseteq V$ for all $i$—for the class of the summary s above, for instance, we have $\lambda(s) = v$ and $\lambda(U_i) = V_i$; we also have

$\lambda(t) = v'$ in case $t \neq \perp$, and $v' = \perp$ otherwise. Let us call such a tuple a *bounded summary*. The idea behind the model-checking algorithm of NT-$\mu$ is that for any formula $\varphi$, we can maintain, symbolically, the set of bounded summaries that satisfy it. Once this set is computed, we can compute the set of bounded summaries for formulas defined inductively in terms of $\varphi$. This computation follows directly from the semantics of the formula; for instance, the set for the formula $\langle loc \rangle \varphi$ contains all bounded summaries $\langle v, v', V_1, \ldots, V_k \rangle$ such that for some $v'' \in V$, we have $v \xrightarrow{loc} v''$, and, letting $V_i''$ comprise the elements of $V_i$ that are reachable from $v''$, $\langle v'', v', V_1'', \ldots, V_k'' \rangle$ satisfies $\varphi$.

Let us now define bounded summaries formally. Consider any state $u$ in an NSM $\mathcal{M}$ with state set $V$. A state $u'$ is said to be the *unmatched call-ancestor state* of state $u$ if there is a node $s$ labeled $u$ in $\mathcal{T}^V(\mathcal{M})$ such that $u'$ is the label of the unmatched call-ancestor of $s$ (we have a predicate $Anc_V(u', u)$ that holds iff this is true). Note that a state may have multiple unmatched call-ancestor states. If there is a node $s$ labeled $u$ in $\mathcal{T}^V(\mathcal{M})$ such that $Anc(s) = \perp$, we set $Anc_V(\perp, u)$.

A state $v$ is a *matching exit state* for a pair $(u, u')$, where $Anc_V(u', u)$, if there are nodes $s, s', t$ in $\mathcal{T}^V(\mathcal{M})$ such that $t \in ME(s)$, $s'$ is the unmatched call-ancestor of $s$, and labels of $s$, $s'$, and $t$ are $u$, $u'$, and $v$ respectively (a pair $(u, \perp)$ has no matching exit state).

The modeling intuition is that from a program state modeled by NSM state $u$ and a stack with a single frame modeled by the state $u'$, control may reach a $u''$ in the same context, and then return at the state $v$ via a transition $(u'', u') \xrightarrow{ret} v$. Using well-known techniques for pushdown models [1], we can compute, given a state $u$, the set of $u'$ such that $Anc_V(u', u)$, and for every member $u'$ of the latter, the set $MES(u, u')$ of matching exit states for $(u, u')$, in time polynomial in the size of $\mathcal{M}$.

Now, let $n$ be the arity of the formula $\varphi$ in whose model-checking problem we are interested. A *bounded summary* is a tuple $\langle u, u', V_1, \ldots, V_k \rangle$, where $0 \leq k \leq n$, $Anc_V(u', u)$ and for all $i$, we have $V_i \subseteq MES(u, u')$. The set of all bounded summaries in $\mathcal{M}$ is denoted by $BS$.

Let $\mathcal{E}_{SL} : Free(\varphi) \rightarrow 2^{BS}$ be an environment mapping free variables in $\varphi$ to sets of bounded summaries, and let $\mathcal{E}_\emptyset$ denote the empty environment. We define a map $Eval(\varphi, \mathcal{E}_{SL})$ assigning a set of bounded summaries to a NT-$\mu$ formula $\varphi$:

- If $\varphi = p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that $p \in \kappa(u)$ and $k \leq n$.
- If $\varphi = \neg p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, V_2, \ldots, V_k \rangle$ such that $p \notin \kappa(u)$ and $k \leq n$.
- If $\varphi = X$, for $X \in Var$, then $Eval(\varphi, \mathcal{E}_{SL}) = \mathcal{E}_{SL}(X)$.
- If $\varphi = \varphi_1 \vee \varphi_2$ then $Eval(\varphi, \mathcal{E}_{SL}) = Eval(\varphi_1, \mathcal{E}_{SL}) \cup Eval(\varphi_2, \mathcal{E}_{SL})$.
- If $\varphi = \varphi_1 \wedge \varphi_2$ then $Eval(\varphi, \mathcal{E}_{SL}) = Eval(\varphi_1, \mathcal{E}_{SL}) \cap Eval(\varphi_2, \mathcal{E}_{SL})$.
- If $\varphi = \langle call \rangle\ \varphi'\{\psi_1, ..., \psi_m\}$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for some transition $u \xrightarrow{call} u''$ of $\mathcal{M}$, we have a bounded summary $\langle u'', u'', V_1', V_2', ..., V_m' \rangle \in Eval(\varphi', \mathcal{E}_{SL})$, and for all $v \in V_i'$, where $i = 1, \ldots, m$, we have $\langle v, u', V_1'', \ldots, V_k'' \rangle \in Eval(\psi_i, \mathcal{E}_{SL})$, where $V_j'' = V_j \cap MES(v, u')$ for all $j \leq k$.
- If $\varphi = [call]\ \varphi'\{\psi_1, ..., \psi_m\}$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for all $u''$ such that there is a transition $u \xrightarrow{call} u''$ in $\mathcal{M}$, we have a bounded summary $\langle u'', u'', V_1', V_2', ..., V_m' \rangle \in Eval(\varphi', \mathcal{E}_{SL})$,

and for all $v \in V_i'$, where $i = 1, \ldots, m$, we have $\langle v, u', V_1'', \ldots, V_k'' \rangle \in Eval(\psi_i, \mathcal{E}_{SL})$, where $V_j'' = V_j \cap MES(v, u')$ for all $j \leq k$.

- If $\varphi = \langle loc \rangle\ \varphi'$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1 \ldots, V_k \rangle$ such that for some $v$ such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap MES(v, u'), \ldots, V_k \cap MES(v, u') \rangle \in Eval(\varphi', \mathcal{E}_{SL})$.
- If $\varphi = \langle loc \rangle\ \varphi'$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1 \ldots, V_k \rangle$ such that for some $v$ such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap MES(v, u'), \ldots, V_k \cap MES(v, u') \rangle \in Eval(\varphi', \mathcal{E}_{SL})$.
- If $\varphi = \langle ret \rangle\ R_i$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that (1) $V_i = \{u''\}$, (2) $\mathcal{M}$ has a transition $(u, u') \xrightarrow{ret} u''$, and (3) for all $j \neq i$, $V_j = \emptyset$.
- If $\varphi = \langle ret \rangle\ R_i$, then $Eval(\varphi, \mathcal{E}_{SL})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for all transitions of the form $(u, u') \xrightarrow{ret} u''$, we have (1) $V_i = \{u''\}$, and (2) for all $j \neq i$, $V_j = \emptyset$.
- If $\varphi = \mu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{SL}) = FixPoint\ (X, \varphi', \mathcal{E}_{SL}[X := \emptyset])$.
- If $\varphi = \nu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{SL}) = FixPoint\ (X, \varphi', \mathcal{E}_{SL}\ [X := BS])$.

Here $FixPoint\ (X, \varphi, \mathcal{E}_{SL})$ is a fixpoint computation function that uses the formula $\varphi$ as a monotone map between subsets of $BS$, and iterates over variable $X$. This computation is as in Algorithm 1:

**Algorithm 1** Calculate $FixPoint\ (X, \varphi, \mathcal{E}_{SL})$

```
  X' ← Eval(φ, E_SL)
  if X' = E_SL(X) then
    return X'
  else
    return FixPoint (X, φ', E_SL[X := X'])
  end if
```

Now we can easily show that for an NSM $\mathcal{M}$ with initial state $v_{in}$ and a closed NT-$\mu$ formula $\varphi$, $\mathcal{T}(\mathcal{M})$ satisfies $\varphi$ if and only if $\langle v_{in} \rangle \in Eval(\varphi, \mathcal{E}_\emptyset)$, and that $Eval(\varphi, \mathcal{E}_\emptyset)$ is inductively computable. To understand this more concretely, let us see how this model-checking algorithm runs on our running example. Consider the NSM abstraction $\mathcal{M}_{foo}$ in Sec. 2.3, and suppose we want to check if a write action is locally reachable from the initial state. The NT-$\mu$ property specifying this requirement is $\varphi = \mu X.(wr \vee \langle loc \rangle X \vee \langle call \rangle \varphi' \{X\})$, where $\varphi' = \mu Y.(\langle ret \rangle R_1 \vee \langle loc \rangle Y \vee \langle call \rangle Y\{Y\})$.

We show how to compute the set of bounded summaries satisfying $\varphi'$—the computation for $\varphi$ is very similar. After the first iteration of the fixpoint computation that builds this set, we obtain the set $\mathbf{S_1} = \{\{\langle v_5, v_2, \{v_2'\}\rangle\}$ (the set of summaries satisfying $\langle ret \rangle R_1$). After the second step, we obtain $\mathbf{S_2} = \mathbf{S_1} \cup \{\langle v_2', v_2, \{v_2'\}\rangle, \langle v_3, v_2, \{v_2'\}\rangle, \langle v_4, v_2, \{v_2'\}\rangle\}$, and the next set computed is $\mathbf{S_3} = \mathbf{S_2} \cup \{\langle v_1, v_2, \{v_2'\}\rangle\}$. Note that in these two steps, we only use local edges in the NSM. Now, however, we have found a bounded summary starting at the "entry state" of the procedure *foo*, which may be plugged into the recursive call to *foo*. More precisely, we have $(v_2, v_1) \in \Delta_{call}$, $\langle v_1, v_2, \{v_2'\}\rangle \in \mathbf{S_3}$, and $\langle v_2', v_2, \{v_2'\}\rangle \in \mathbf{S_3}$, so that we may now construct $\mathbf{S_4} = \mathbf{S_3} \cup \langle v_2, v_2, \{v_2'\}\rangle$. This ends the fixpoint computation, so that $\mathbf{S_4}$ is the set of summaries satisfying $\varphi'$.

Let us now analyze the complexity of this algorithm. Let $N_V$ be the number of states in $\mathcal{M}$, and let $n$ be the arity of the formula in question. Then the total number of bounded summaries in $\mathcal{M}$ that we need to consider is bounded by $N = N_V^2 2^{N_V n}$. Let us now assume that union or intersection of two sets of summaries, as well as membership queries on such sets, take linear time. It is easy to see that the time needed to evaluate a non-fixpoint formula $\varphi$ of arity $n \leq |\varphi|$ is bounded by $O(N^2|\varphi|N_v)$ (the most expensive modality is $\langle call \rangle \varphi' \{\psi_1, \ldots, \psi_n\}$, where we have to match an "inner" summary satisfying $\varphi'$ as well as $n$ "outer" summaries satisfying the $\psi_i$-s). For a fixpoint formula $\varphi$ with one fixpoint variable, we may need $N$ such evaluations, so that the total time required to evaluate $Eval(\varphi, \mathcal{E}_\emptyset)$ is $O(N^3|\varphi|N_V)$. For a formula $\varphi$ of alternation depth $d$, this evaluation takes time $O(N^{3d} N_V^d \ |\varphi|)$, i.e., exponential in the sizes of $\mathcal{M}$ as well as $\varphi$.

It is known that model-checking alternating reachability specifications on a pushdown model is EXPTIME-hard [24]. It is not hard to generate a NT-$\mu$ formula $\varphi$ from a $\mu$-calculus formula $f$ expressing such a property such that (1) the size of $\varphi$ is linear in the size of $f$, and (2) $\mathcal{M}$ satisfies $\varphi$ if and only if $\mathcal{M}$ satisfies $f$. It follows that model-checking a closed NT-$\mu$ formula $\varphi$ on an NSM $\mathcal{M}$ is EXPTIME-hard. Combining, we conclude that model-checking a NT-$\mu$ formula $\varphi$ on an NSM $\mathcal{M}$ is EXPTIME-complete. Better bounds may be obtained if the formula has a certain restricted form. For instance, it can be shown that for linear time (Büchi or reachability) requirements, model-checking takes time polynomial in the number of states of $\mathcal{M}$. The reason is that in this case, it suffices to only consider bounded summaries of the form $\langle v, v', \{v''\}\rangle$, which are polynomial in number. The fixpoint computation stays the same.

Note that our decision procedure is very different from known methods for branching-time model-checking of pushdown models [24,12]. The latter are not really implementable; our algorithm, being symbolic in nature, seems to be a step in the direction of practicality. An open question here is how to represent sets of bounded summaries symbolically. Also, note that our algorithm directly implements the operational semantics of NT-$\mu$ formulas over bounded summaries. In this regard NT-$\mu$ resembles the modal $\mu$-calculus, whose formulas encode fixpoint computations over sets; to model-check $\mu$-calculus formulas, we merely need to perform these computations. Unsurprisingly, our procedure is very similar to classical symbolic model-checking for the $\mu$-calculus.

## References

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems*, 27(4):786–818, 2005.

[2] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*, 2006.

[3] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Computer-Aided Verification, CAV'06*, 2006.

[4] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04: Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, LNCS 2988, pages 467–481. Springer, 2004.

[5] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *Automata, Languages and Programming: Proceedings of the 32nd ICALP*, LNCS 3580, pages 1102–1114. Springer, 2005.

[6] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, pages 202–211, 2004.

[7] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, 2006.

[8] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[9] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.

[10] T. Ball and S. Rajamani. The SLAM toolkit. In *Computer Aided Verification, 13th International Conference*, 2001.

[11] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, R. Leino, and E. Poll. An overview of JML tools and applications. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems*, pages 75–89, 2003.

[12] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[13] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft, Available at `http://www.grappa.univ-lille3.fr/tata/`, 2002.

[14] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, pages 232–241, 2006.

[15] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.

[16] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[17] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[18] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[19] T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[20] D.A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 68–78, 1998.

[21] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[22] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[23] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *IEEE Symp. on Security and Privacy*, pages 52–63, 1998.

[24] I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, 2001.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*

# Specifying, Relating and Composing Object Oriented Interfaces, Components and Architectures [1]

Manfred BROY
*Institut für Informatik, Technische Universität München*
*D-80290 München Germany, broy@in.tum.de*
*http://wwwbroy.informatik.tu-muenchen.de*

**Abstract.** We introduce an abstract theory for systems, components, composition, architectures, interfaces, and compatibility. We apply this theory to object orientation and work out an instance of that theory that covers these notions in terms of a formal model of objects, classes, components, and architectures as well as of interfaces of classes and components and their specification. We define and analyze, in particular, interfaces for object oriented software systems and their architectures. We deal with "design by contract" as well as "specification by contract" and analyze their limitations. We show how to specify these interfaces by logical formulas in the style of specification by contract, by state machines and also by interaction diagrams. We treat composition given a formal definition of class composition and analyze semantic complications. We discuss, in particular, how we can extend concepts from object orientation towards components and more sophisticated ways to handle interfaces and hierarchical architectures. Our approach is based on the concept of states, state assertions, and state machines. A pragmatic goal is to explore simple and tractable ways to describe interfaces.

## 1. Motivation

Software development is today, without any doubts, one of the most difficult and significant tasks in the engineering of complex systems. Modern software systems typically are *deployed* and *distributed* on large networks; they are *dynamic*, and accessed *concurrently* via a couple of independent user interfaces. They are based on software infrastructures such as operating systems and middleware, which offers the communication services of object request brokers.

To master complexity in the development process large software systems are typically built in terms of architectures in a modular fashion and hierarchically structured into components. These components are grouped together into software architectures. These ideas of structuring software go back to "structured programming" according to Dijkstra, Hoare, Dahl, Wirth, and, in particular, to Parnas (see [16]). We apply their ideas to object oriented systems. We start our discussion in terms of the idea of assertions and

[1] This work was done in cooperation with and was partially sponsored by sd&m AG.

"design by contract" to specify methods. We then show shortcomings and limitations of design by contract specifications and explore ways to overcome those.

Basically, we treat the following basic concepts to specify the behaviour of classes and objects following the idea of interfaces:

- Predicates on the interaction between the objects, classes, and components in terms of streams of invocations and return messages
  - Message sequence charts (interaction charts)
- State based specifications
  - Pre/post assertion specifications
  - State machines

In the following we discuss these approaches, their advantages as well as their limitations. We underline and illustrate our discussions by characteristic examples.

In this paper we try to develop a formal approach to modular interface specification of classes and components and their composition. We deal with the following issues and concepts

- methods and their specification by contract
- classes (with simple "one-way" export interfaces) and their specification by
  - contract
  - state machines
  - stream processing functions
- a modular view of classes taking also into account methods of classes that are used (in forwarded method invocations) leading to import/export interfaces.

This modular view is badly needed when trying to compose classes to large more complex systems. The architectural decomposition and the systematic integration needs interface specifications.

We introduce and discuss the concept of components, which are generalizations of classes that have several export/import interfaces. Our basic ideas read as follows:

- A basic class is a simple form of a component with only one export interface.
- A generalization yields families of classes with interfaces with export and import parts.
- A component is a generalization of a class; it features several interfaces with export and import parts.
- The composition of components is realized by matching their export/import interfaces.
- Export/import interfaces are mandatory to deal with composition of components.
- Export/import interfaces introduce a number of severe complications
  - Simple pre/post specifications of method calls do no longer work. Method calls are carried out by a sequence of forwarded calls and returns, in general.
  - We have to make call stacks explicit to handle return messages of forwarded calls effectively.
- In the composition of components that are represented by state machines we obtain state machines that have internal state transitions.

In the following we discuss all these problems, concepts, and demonstrate all the mentioned complications and obstacles by examples in detail and show solutions. Throughout the paper we do not treat inheritance, at all.

## 2. Interfaces and Compatibility: A Theory of Substitutability, Modularity, and Observability

In this section we fix the essential notions of abstraction, interface and architecture. We describe the essentials of a basic theory (see also [6]).

### *2.1. Syntactic Framework*

We assume some syntax to describe components and architectures. This means that we have a syntactic notion of components. Let LC be the formal language of components. Every syntactic element $c \in LC$ represents a component. For our purposes, it does not matter whether we describe components by logics (as in [3]), graphical languages (like UML) or by textual languages (like programming languages such as Java or C# or by architecture description languages).

In addition, we assume a *syntactic composition operator*. It is a partial operation on components that allows us to compose two components:

$$\otimes: LC \times LC \rightarrow LC \qquad \text{written in infix notation}$$

Here "partial" means that not for every pair c1, c2 ∈ LC of components the operation yields a well-defined result. Only if two components c1, c2 fit together with respect to their syntactic properties (their "syntactic interfaces" given by the types of their shared variables, their messages or method calls) their composition is meaningful. For every pair c1, c2 ∈ LC of components, c1 ⊗ c2 is called *composed component*, if for c1 and c2 their composition is defined.

In order to deal with the partiality of composition we assume a relation

$$\Re: \wp(LC) \rightarrow \mathbb{B}$$

$\Re(C)$ holds for a set of components $C \subseteq LC$ if their composition is well-defined. So if and only if $\Re(\{c1, c2\})$ holds for components $c1, c2 \in LC$ (for simplicity we ignore here the case $c1 = c2$ and assume that $c1 \neq c2$ holds) we get that $c1 \otimes c2$ yields a well-defined result (we assume that $c \otimes c$ is not well-defined; in fact, this is not a real restriction, since we may assume that all components have at least different names).

For finite sets of components $\{c_1, ..., c_k\} \subseteq LC$ with $\Re(\{c_1, ..., c_k\})$ we define composition as follows

$$\prod \{c_1, ..., c_k\} = c_1 \otimes ... \otimes c_k$$

This notation is justified by assuming that the operator ⊗ is commutative and associative. Whenever for a set of components $C \subseteq LC$ the proposition $\Re(C)$ holds, the term $\prod C$ is called an *architecture* with components from C.

Using the operation $\otimes$ we get a hierarchical concept of components – composing two components yields a component, again. In fact, every architecture represents again a component. A more restricted concept is obtained, if $\prod C$ is not seen as a component again. Then hierarchical design is not supported. However, such a restriction is not substantial – at least in theory. Given two (disjoint) sets of components $C1, C2 \subseteq LC$ we easily define $\prod C1 \otimes \prod C2$ by $\prod(C1 \cup C2)$. We loose the hierarchy, however, this way.

To keep our framework simple, we only introduced the concept of components here but not that of connectors as found in some architecture description languages. Connectors are easily subsumed by modelling them as special versions of components.

A system is a component with specific properties. We assume that in the set of all components a subset $LS \subseteq LC$ is given that characterizes comprehensive self-contained systems.

### 2.2. *Semantic Framework: Substitutability and Compatibility*

When dealing with specifications and behaviours we are in particular interested in an essential semantic relation for components namely *substitutability* (see [22,15]).

**Definition.** Substitutability and Compatibility
A component c1 is called *substitutable* for a component c2 if the following holds: in every system that is syntactically correct and in which c2 occurs as a component we can replace the component c2 by c1 and this results in a system that is again syntactically correct and the observable behaviour it shows is identical to (or a refinement of) the observable behaviour of the original system. In this case we also say that component c1 is compatible to (or refined by) component c2. □

This definition is informal, since it does not provide a formal model of observable behaviour. The concept of substitutability is closely related to that of interface specifications, as we will show in more detail below. Each interface specification has to characterize the set of components that can be used as replacements for the specified component. Thus an interface specification for a component defines a set of compatible components. We will discuss whether an interface specification can be seen as an "abstract" component.

The essential concept that formalizes substitutability is observability. Looking at an entity from the outside we can observe certain actions and events triggering state changes. By such observations we filter out the relevant information about systems. If we restrict the concept of observations we obtain a more abstract view.

### 2.3. *Observability and Compatibility*

We give a more formal approach to observability in the following sections. We distinguish syntactic and semantic issues.

#### 2.3.1. *Syntactic Compatibility*

Our concept of syntactic composability formalized by the predicate $\mathfrak{R}$ introduces the idea of *syntactic compatibility* of components. Given two components $c1, c2 \in LC$, the component c2 is called syntactically compatible to c1, if we can use component c2 whenever we use c1 without running into syntactic errors.

To formalize this, we introduce a relation

$$\triangleright \subseteq LC \times LC$$

with the following definition for components $c1, c2 \in LC$

$$c1 \triangleright c2 \Leftrightarrow \forall C \subseteq LC \backslash \{c1, c2\}: \Re(C \cup \{\text{c1}\}) \Rightarrow \Re(C \cup \{\text{c2}\})$$

The proposition $c1 \triangleright c2$ expresses that whenever component c1 can be used as a component in a system (leading to a syntactically correct system), c2 can be used instead, too. Of course, the system that we obtain by replacing component c2 for c1 may show a different behaviour. We only require that the resulting systems be syntactically well formed.

Syntactic substitutability induces an equivalence relation

$$\sim \;\subseteq LC \times LC$$

on the set of components that corresponds to the idea of mutually syntactically substitutability. This relation is easily defined as follows:

$$c1 \sim c2 \Leftrightarrow (c1 \triangleright c2 \wedge c2 \triangleright c1)$$

and called *syntactic equivalence*.

### 2.3.2. *Observable Equivalence*

After having introduced a basic syntactic framework of components and architectures, we develop a more semantic view onto components. This cannot be done without a precise semantic perspective. We introduce such a concept only for systems, to begin with, by assuming that we have some idea of observations at least about systems, which are elements in the set LS.

To formalize *observable equivalence* we are interested in the question, under which conditions two systems are observably equivalent. Syntactic equivalence was introduced above. Observable equivalence is modelled by an equivalence relation on systems.

$$\cong \;\subseteq LS \times LS$$

The equivalence relation expresses by the proposition $s1 \cong s2$ that two systems $s1, s2 \in LS$ are observably and thus semantically equivalent.

We can also be more concrete and introduce a concept of observation explicitly. Let OBS be the set of observations about a system and

$$\text{Obs}: LS \rightarrow OBS$$

be the functions that maps systems onto observations. We assume:

$$s1 \cong s2 \Leftrightarrow Obs(s1) \cong Obs(s2)$$

for all systems $s1, s2 \in LS$.

Theoretically and practically, there are of course many options to define observability and observable equivalence. In the end, observability has to be related to the usersŠ expectations and views onto a system making explicit which observations about a system are relevant for the users. Practically, what is a good notion of observation for systems seems often obvious. In principle, however, we may include also non-functional aspects into observability such as reaction time or consumed resources. In the following, we are interested exclusively in observability in terms of functional ("behavioural") properties.

### *2.3.3. Refinement*

Another way to introduce observability is to introduce a partial order $\succ$ on systems that defines observability such $s1 \succ s2$ expresses that all observations about s2 or also observations about s1. On components we introduce the relation of semantic substitutability for components. We call this relation *refinement* and denote it by the relation

$$\succ \ \subseteq LC \times LC$$

The relation $\succ$ is assumed to be transitive and reflexive. Semantic substitutability for components has to be and can be directly related to observable equivalence of systems. In fact, we could define the relation $\succ$ formally based on the observability relation $\cong$. We rather keep the two relations independent to begin with and show, how they are and must be related then.

## *2.4. Compositionality and Modularity*

Refinement for components has to be consistent with composition. This is called *compositionality*. With the introduced concepts we can formally define compositionality of refinement $\succ$ with respect to the observability relation $\cong$.

**Definition.** Compositionality and modularity
The relation $\succ$ is called *compositional* (or *modular*) with respect to the relation of observable equivalence $\cong$, if for all components $c1, c2 \in LC$ we have:

$$c1 \succ c2 \Rightarrow \forall C \subseteq LC \backslash \{\text{c1}, \text{c2}\}, s \in LS:$$
$$\Re(C \cup \{\text{c1}\}) \wedge s \cong \prod(C \cup \{\text{c1}\}) \Rightarrow \Re(C \cup \{\text{c2}\}) \wedge s \cong \prod(C \cup \{\text{c2}\}) \qquad \Box$$

This definition expresses that if $c1 \succ c2$ holds we can replace in any system s that uses the component c1 the component c1 by c2 and get an observably equivalent system.

### *2.4.1. Denotational Semantics*

The core idea of denotational semantics is the definition of the meaning of a programming language in terms of mathematical denotations. Let us denote by MD the set of mathematical denotations. The mapping

$$\beta: LC \to MD$$

maps components onto their denotations (that are mathematical representations of the meaning of the system, component, or program). Moreover we assume a form of composition on the set MD of denotations

$$* : MD \times MD \to MD \qquad \textit{written in infix notation}$$

which is again a partial operation. The key property and idea of denotational semantics is captured by the following equation for $c1, c2 \in LC$ with $\Re(\{c1, c2\})$ :

$$\beta(c1 \otimes c2) = \beta(c1) * \beta(c2)$$

This is called compositionality and is the core concept of denotational semantics.

Given a concept of observations, we call the denotational semantics represented by the mapping $\beta$ consistent with the observation, if there is a mapping

$$\alpha : MD \to OBS$$

where for all $s \in LS$:

$$\alpha(\beta(s)) = Obs(s)$$

In other words, we require that we can obtain the observations about a system from its denotation – the denotational semantics carries enough information to derive all observations from it.

### 2.4.2. *Full Abstractness of Denotational Semantics*

A consistent and compositional denotational semantics is easy to provide: take for the set of denotations MD simply the syntactic set LC of components, for $\beta$ the identity and for $*$ the operation $\otimes$. Then $\alpha$ is easily represented by Obs and we have all the required properties.

However, this merely syntactic construction is not what we are looking for. Abstraction is what we are interested in. We are interested in a more abstract denotational semantics. A denotational semantics represented by $\beta$ is called fully abstract, if it is consistent, compositional and if there does not exist a more abstract denotational semantics that is consistent and compositional, too.

Given two denotational semantics

$$\beta_1 : LC \to MD_1$$

$$\beta_2 : LC \to MD_2$$

we call $\beta_2$ as least as abstract than $\beta_1$, if there is a mapping

$$\gamma : MD_1 \to MD_2$$

such that for all components $c \in LC$:

$$\gamma(\beta_1(c)) = \beta_2(c)$$

This means that there may be components $c1, c2 \in LC$ such that we have

$$\beta_1(c1) \neq \beta_1(c2) \wedge \beta_2(c1) = \beta_2(c2)$$

and moreover

$$\beta_1(c1) = \beta_1(c2) \Rightarrow \beta_2(c1) = \beta_2(c2)$$

for all $c1, c2 \in LC$.

*2.4.3. Full Abstractness of Refinement*

By refinement $\succ$ we can extend the relation $\cong$ from systems to components $c1, c2 \in LC \backslash LS$ by the definition

$$c1 \cong c2 \Leftrightarrow (c1 \succ c2 \wedge c2 \succ c1)$$

This defines what it means that two components and two systems are observably equivalent. For systems $c1, c2 \in LS$, the formula is a straightforward theorem.

We expect that observable equivalence implies syntactic equivalence

$$c1 \cong c2 \Rightarrow c1 \sim c2$$

If the relation $\succ$ is compositional, then for all components $c1, c2, c3, c4 \in LC$ we have

$$\Re(\{\text{c1, c2}\}) \wedge c1 \cong c3 \wedge c2 \cong c4 \Rightarrow \Re(\{\text{c3, c4}\}) \wedge c1 \otimes c2 \cong c3 \otimes c4$$

In this case we call the relation $\cong$ compositional, too, and we speak of a *modular* theory of components and architectures.

Refinement and substitutability is, of course, related to inheritance. Actually, refinement is the semantically more appropriate idea of inheritance – a relation, which in object oriented languages, where inheritance is often just code reuse, is not always guaranteed.

Finally we consider the notion of what it means that refinement is fully abstract.

**Definition.** Full abstractness
The relation $\succ$ is called *fully abstract* for the observability equivalence relation $\cong$, if for all components $c1, c2 \in LC$ we have

$$\begin{aligned} c1 \succ c2 \Leftarrow [\forall C \subseteq LC\backslash\{\text{c1, c2}\}, s \in LS{:}\Re(C \cup \{\text{c1}\}) \wedge s \cong \textstyle\prod(C \cup \{\text{c1}\}) \Rightarrow \\ \Re(C \cup \{\text{c2}\}) \wedge s \cong \textstyle\prod(C \cup \{\text{c2}\})] \end{aligned}$$ □

Full abstractness means that the refinement relation on components is the most abstract relation that guarantees modularity for the chosen concept of observability onto systems.

There is a way to introduce refinement $\succ$ based on the observability relation $\cong$ such that it is always fully abstract. This is achieved by taking the following formula as a definition of refinement:

$$\begin{aligned} c1 \succ c2 \Leftrightarrow [\forall C \subseteq LC\backslash\{\text{c1, c2}\}, s \in LS{:}\Re(C \cup \{\text{c1}\}) \wedge s \cong \textstyle\prod(C \cup \{\text{c1}\}) \Rightarrow \\ \Re(C \cup \{\text{c1}\}) \wedge s \cong \textstyle\prod(C \cup \{\text{c2}\})] \end{aligned}$$

If we introduce $\succ$ independently, then full abstractness is not guaranteed. If $\succ$ is fully

abstract and compositional, however, this formula obviously holds and we have then also for all components c1 and c2 the validity of the following formula:

$$c1 \cong c2 \Leftrightarrow [\forall C \subseteq LC \backslash \{c1, c2\}, s \in LS: \\ (\Re(C \cup \{c1\}) \wedge s \cong \prod(C \cup \{c1\})) \Leftrightarrow (\Re(C \cup \{c2\}) \wedge s \cong \prod(C \cup \{c2\}))]$$

The relations $\cong$ and $\succ$ are called *fully abstract*, if for all components $c1, c2 \in LC$ we have

$$[\forall c \in LC{:}\Re(\{c1, c\}) \Rightarrow c1 \otimes c \cong c2 \otimes c] \Rightarrow c1 \cong c2$$

and respectively

$$[\forall c \in LC{:}\Re(\{c1, c\}) \Rightarrow \Re(\{c1, c\}) \wedge c1 \otimes c \succ c2 \otimes c] \Rightarrow c1 \succ c2$$

Full abstractness is an essential methodologically concept, since it only allows us to replace a component by any of its refinements.

### 2.5. Component and System Interfaces and Specifications

For practical purposes it is difficult to work with an abstract notion of refinement and observable equivalence. It is better and from a methodological point of view more interesting to introduce explicitly the concept of syntactic and semantic interfaces that characterize sets of components that can be used for a certain system. Interfaces are specifications of components.

#### 2.5.1. Interface Specification

The notion of a syntactic interface is straightforward. A syntactic interface defines a set of components. Formally, an interface specification is nothing but a predicate of the form

$$\mathfrak{F}{:}LC \to \mathbb{B}$$

that is closed under the syntactic substitutability relation $\triangleright$. It characterizes a set of components. Actually, then for all components $c1, c2 \in LC$ we have

$$\mathfrak{F}(c1) \wedge c1 \triangleright c2 \Rightarrow \mathfrak{F}(c2)$$

In other words, a syntactic interface $\mathfrak{F}$ characterizes a set of components such that with every components that is syntactically fine with respect to $\mathfrak{F}$ all its valid syntactic replacements do also fulfil $\mathfrak{F}$.

Formally, a semantic interface is a predicate

$$\mathfrak{F}{:}LC \to \mathbb{B}$$

that is closed under the semantic substitutability relation $\succ$. Then for all components $c1, c2 \in LC$ we assume

$$\mathfrak{F}(c1) \wedge c1 \succ c2 \Rightarrow \mathfrak{F}(c2)$$

In other words, a semantic interface $\mathfrak{F}$ characterizes a set of components such that with every component that is semantically correct with respect to $\mathfrak{F}$ all its valid refinements do also fulfil $\mathfrak{F}$.

A semantic interface $\mathfrak{F}$ characterizes, in general, a set of components. Sometimes there is a concrete representation of a semantic interface by a component $c \in LC$ such that for all components $c' \in LC$ we have

$$\mathfrak{F}(c') \Leftrightarrow c \succ c'$$

Then $\mathfrak{F}$ is called a *concrete* interface, otherwise an *abstract* interface.

Logical implication induces a refinement relation on interfaces. This way notions such as compositionality or full abstractness carry over to interfaces.

A consequent methodological step is to consider abstract interface specifications as non-operational components, too. Then in architectures specifications and realized components can be freely combined. We work that out in the following section.

### 2.5.2. *Composing Specifications*

There is, in the general case, not a one-to-one correspondence between specifications and components. More precisely, not every interface specification is concrete, in general. A specification may be fulfilled by many components and a component may fulfil many specifications. Nevertheless, we may be interested to "lift" the notion of composition and refinement to specifications. Let CS denote the set of all specifications. To do that we introduce ("extend") the composition operations to specifications

$$\otimes : CS \times CS \to CS \qquad \textit{infix}$$

where for specifications $p1, p2 \in CS$ we get (whenever $p1 \otimes p2$ is well-defined)

$$(p1 \otimes p2)(c) \equiv \exists c1, c2 \in LC : c = c1 \otimes c2 \wedge p1(c1) \wedge p2(c2)$$

In other words we mimic the notion of compositions at the language level at the specification level. The same applies for refinement:

$$(p2 \Rightarrow p1) \Leftrightarrow \forall c, c' \in LC : p2.(c) \wedge c \succ c' \Rightarrow p1.(c')$$

These notions of composition and refinement are properly reflected at the level of specifications.

## 2.6. *Architectures*

Systems architectures are given by sets of components and a description, how these are composed such that the composition is well-defined. Architectures are given by terms of compositions of components.

Architecture specifications consist of interface specifications of the set of components (of abstract or concrete components) and a description, how these are composed where we require that the composition is well-defined. From a more practical point of view, we introduce names for the interface specifications that represent components and additional information "how" the composition connects the components.

### 2.7. Final Remarks on the Theory

An instance of the theory that offers all the concepts introduced so far and fulfils all the given rules can be found in the approach Focus (see [3]). In contrast, such a fully worked-out theory does not exist so far for object orientation.

What we have described in this section is essential for a theory and methodology for the specification and modular design of architectures and their components. Of course, the theory alone is not enough for doing engineering. Obviously, we need, in addition, a useful syntax to represent interface specifications and architectures. The theory, however, provides a theoretical framework that gives hints which properties an approach with a concrete syntax has to fulfil.

A challenge is, however, to work out an explicit denotational semantics for a given language, with all the relations introduced as well as a concrete syntax for writing interface specifications.

## 3. Components and Interfaces in Object Orientation

In this section we study the object oriented programming paradigm. We introduce the essential syntactic and semantic notions of *method, method specification, interface,* and class and finally that of a *component*. We analyze ideas of design by contract (see [12]).

We introduce the notion of a method, interface, class and component based on the idea of design by contract, on state machines, and that of a data stream and components interacting by exchanging streams of data. Throughout this paper we work with only a few basic notations for state machines and data streams. On this basis we discuss a semantic theory for object orientation.

### 3.1. Methods and Invocation/Return Messages

In this section we show an approach to interfaces and components based on ideas used in object oriented software development.

#### 3.1.1. Types and Methods

We work with interfaces that refer to the concept of data types. We deal with variable types and constant types. A constant type is basically a set of data values.

**Definition.** Types A type is either a *constant* type or a *variable* type. Constant types are basically sets of data values. An identifier with constant type denotes a value of that set. A variable type is denoted by **Var** T where T is a constant type. An identifier with variable type denotes a variable (in object orientation called an attribute) that has assigned a value of the set of elements of type T. Every class name is also a type. □

A method in object orientation consists syntactically of its method header and its method body. Since we are not interested in the design of algorithms as done in programming in the small in this paper and thus not in the particular code forming the method body we just deal with syntactic method headers in the following. We assume a special set Object of object identifiers. Object identifiers are typed. Their types are represented by their class names.

**Definition.** Method header
A method header has the syntactic form (for simplicity we only consider the special case, where the method has one constant and one variable as formal parameters, the generalisation of our notions to many parameters is straightforward)

**method** m (w : WT, v : **Var** VT)

where w and v are identifiers for parameters and WT as well as **Var** VT are their types. Identifiers with constant types carry input and those with variable types serve for output (carry results).

The set of method invocations INVOC(m) for the method m is defined by the following equation:

INVOC(m) = { m(c1, c2, w, v, v'): w ∈ WT, v, v' ∈ VT, c1, c2 ∈ Object}

where the phrase $d \in T$ expresses that d is a value of type T and m(b1, b2, w, v, v') denotes a tuple of values. Here c1 denotes the *caller* and c2 the *callee* of the method invocation. □

For simplicity we do not distinguish between input parameters, transient parameters, and result parameters. Transient parameters carry both input and output for the method invocation. Also for simplicity, we do not consider the question, whether the callee is an object of the appropriate type – more precisely an object of a class that exports the method m, but take that for granted.

In specification by contract we treat method invocations as atomic state changes. Later we treat method invocations as sequences of state changes, starting with the method invocation and ending with its corresponding method return message. In the later case, method invocations are represented by two messages.

### *3.1.2. Specification by Contract*

A method can be specified by contract as long as we can understand it as a definition of an atomic state change. To do that, we have to refer to the states of an object or more precisely to the states of an object oriented system before and after the invocation of a method.

**Definition.** States and their Attributes
The states of the objects of a class are determined by the valuations of the attributes of that class. An attribute is a typed identifier. An attribute set V is a set of the form

$$V = \{a_1 : T_1, \dots , a_n : T_n\}$$

where $a_1, \dots, a_n$ are (distinct) identifiers and $T_1, \dots, T_n$ are their types. A valuation of the attribute set V is a mapping

$$\sigma : V \to UD$$

where UD is the universe of data values. Of course, we assume for each valuation $\sigma$ that for each attribute a the value $\sigma(a)$ has the type given to the attribute. $\sigma$ is also called a state of V. By $\Sigma(V)$ we denote the set of all states for V. □

Given the concept of a state of attributes and objects we now define what it means to write a specification by contract for a method.

**Definition.** Specification by contract for a Method
Let V = {a : T} be an attribute set (for simplicity of notation we consider again only the simple case with just one attribute – the generalization to n attributes is quite straightforward). A *specification by contract* for a method with header

**method** m (w : WT, v : **Var** VT)

in a class with attribute set V is given by the scheme

**method** m (w : WT, v : **Var** VT)
  **pre** $P(a, w, v)$
  **post** $Q(a, w, v, a', v')$

In this scheme

$P(a, w, v)$

and

$Q(a, w, v, a', v')$

denote predicates – more precisely, formulas in predicate logic called *assertions* which contain the identifiers $a, w, v$ and $a, w, v, a', v'$ as their free variables. We assume that each "primed" variable $a', v'$ denotes the value of that variable in the state after the termination of the invocation. □

If

$$P(a, w, v) \Rightarrow \exists a', v' : Q(a, w, v, a', v')$$

does not hold then we can enhance the precondition by

$$P(a, w, v) \wedge \exists a', v' : Q(a, w, v, a', v')$$

We give a first simple example for a specification by contract. We choose a method from a class defining lists. We consider the class of lists later and we select only one method here. To deal with lists we have to have a mathematical definition of the data structure of sequences underlying the idea of a list. For that purpose we use the following algebraic specification as a basis defining the types of our examples:

**SPEC** SEQ =
{ **based_on** BOOL,
 **type** Seq $\alpha$,
 $\langle\rangle$ : Seq $\alpha$, "empty sequence"
 $\langle_\rangle$ : $\alpha \rightarrow$ Seq $\alpha$, Mixfix "one-element sequence"
 $\circ$ : Seq $\alpha$, Seq $\alpha \rightarrow$ Seq $\alpha$, Infix "concatenation"
 iseseq : Seq $\alpha \rightarrow$ Bool,
 first, last : Seq $\alpha \rightarrow \alpha$,
 head, rest : Seq $\alpha \rightarrow$ Seq $\alpha$,

 index: $\alpha$, Seq $\alpha \rightarrow$ Nat,

length: Seq $\alpha \rightarrow$ Nat,
ith : Nat, Seq $\alpha \rightarrow \alpha$,
drop : $\alpha$, Seq $\alpha \rightarrow$ Seq $\alpha$,
cut: Seq $\alpha$, Nat, Nat $\rightarrow$ Seq $\alpha$

Seq $\alpha$ **generated_by** $\langle\rangle$, $\langle_\rangle$, $^\circ$ ,

iseseq($\langle\rangle$) = true,
iseseq($\langle a\rangle$) = false,
iseseq($x^\circ y$) = and(iseseq(x), iseseq(y)),

length($\langle\rangle$) = 0,
length($\langle$ a $\rangle$) = 1,
length($x^\circ y$) = length(x) + length(y),

ith(1, $\langle$ a $\rangle^\circ y$) = a,
ith(n+1, $\langle$ a $\rangle^\circ y$) = ith(n, y),

index(a, $\langle\rangle$) = 0,
index(a, $\langle a\rangle$) = 1,
$a \neq b \Rightarrow$ index(a, $\langle b\rangle^\circ x$) = **if** index(a, x) = 0 **then** 0 **else** 1 + index(a, x) **fi**

drop(a, $\langle a\rangle^\circ x$) = x,
$a \neq b \Rightarrow$ drop(a, $\langle b\rangle^\circ x$) = $\langle b\rangle^\circ drop(a, x)$,

cut(s, i, 0) = $\langle\rangle$
cut(s, 0, j+1) = $\langle first(s)\rangle^\circ$ cut(rest(s), 0, j)),
cut(s, i+1, j+1) = cut(rest(s), i, j),

$x^\circ\langle\rangle$ = x = $\langle\rangle^\circ x$,
$(x^\circ y) \circ z = x \circ (y^\circ z)$,

first($\langle a\rangle^\circ x$) = a,
last($x^\circ\langle a\rangle$) = a,
head($\langle a\rangle^\circ x$) = $\langle a\rangle$,
rest($\langle a\rangle^\circ x$) = x

}

We do not go deeper into details of the algebraic specification of data structures but rather refer to [23]. Throughout this paper we use algebraic specification only as an auxiliary technique to specify the data types which we refer to in specifications of class behaviours.

**Example.** Specification by contract (see [12])
The following section gives a syntactic interface of the class List. We consider only one method here and assume only one attribute

v : **Var** Seq Data

We give the following example of specification by contract for a method that gets access ("reads") the ith element of sequence v:

**method** get (i : Nat, r : **Var** Data);
**pre** $1 \leq i \leq length(v)$
**post** $r' = ith(i, v) \wedge v' = v$

It is essential to write also v' = v to express that the attribute v is not changed by the execution of the method invocation. □

It is important to emphasize that the specification by contract approach requires knowledge about the local state structure, determined by the attribute names and their types, of the respective class and object.

In our post-conditions we use a simple notational convention: to avoid a lot of assertions of the form $a' = a$, which state that attribute a is not changed, we assume that an attribute a (or a variable parameter) is not changed, i.e. that $a' = a$ holds in the post-condition, if the identifier a' does not occur syntactically in the post-condition.

### *3.2. Simple Export Interfaces*

We start with interfaces of conventional classes, which we call pure *export classes and resp. export interfaces.* An object oriented export interface is described simply by a collection of (exported) class names and the methods offered by them. The description of a syntactic class interface obviously is simple. It is the collection of a set of syntactic method headers for each of the classes. Nothing is said about behavioural aspects.

Of course, we gain more flexibility, if we consider also sets of objects as part of the interface. Since we are rather interested in foundational issues, we do not do that. Nevertheless, the approach can schematically be extended into this direction.

#### *3.2.1. Simple Object Oriented Interfaces*

We start with interfaces of conventional classes which we call "export". An "export" object oriented interface consists simply of a set of class names and of a collection of methods M (more precisely a set of methods for each class). For simplicity, we do not capture in our formalism which of the class names offers which methods, which of course could be easily formalized, too. A simple class defines and provides only such interfaces.

**Example.** Syntactic List interfaces
The syntactic interface of the example class List is given by the following set of methods:

```
interface SynList {
        method add (x : Data);
        method size (s : Nat);
        method get (i : Nat, r : Var Data);
        method contains (x : Data, r : Var Bool);
        method indexOf (x : Data, r : Var Nat);
        method remove (x : Data);
}
```

This example defines just a *syntactic* interface by listing a set of method headers. It specifies nothing, however, about the effects or behaviour of these methods. □

The description of a syntactic class interface obviously is simple. It is the collection of a set of syntactic method headers. Nothing is said about behavioural aspects.

**Definition.** Syntactic export interface
A *syntactic export interface* consists of a name (the interface name) and a set of class names (used as types) and for the class names the set M of method headers. We assume for simplicity that all methods have different names, since we do not want to deal with overloading. We also assume that each method is related to a class. By

$$\text{INVOC}(M) = \bigcup_{m \in M} \text{INVOC}(m)$$

we denote the set of all possible invocations of methods that are in the syntactic export interface M. □

In the following we discuss semantic, behavioural notions of such interfaces. Such information is needed if we plan to use the classes without wanting to look at their code.

*3.2.2. Specification by Contract for Methods in Export Interfaces*

In this section we show how to express specifications of classes by contract. It is essentially based on specifying methods by contract as introduced above.

**Definition.** State transition assertion
Given a set of attributes V = {a : T} a *state transition assertion* is an assertion of the form

$$R(a, a')$$

that restricts the state changes and also the set of reachable states. If the primed attribute $a'$ does not occur in the assertion, we speak of a *state assertion* (and also of an invariant for a class), otherwise of a *state transition assertion*. □

We use state transition assertions and state assertions to provide behavioural specifications for classes in addition to the assertions given for the individual methods in the specification by contract.

Now we give the definition of the specification of a class by contract.

**Definition.** Specification of classes by contract
For a *syntactic interface* consisting of a set of method headers a specification by contract is given by a set of typed attributes defining the class state and a specification by contract for each of its methods.
In addition, a state invariant may be given by the construct

**invariant** Inv(a)

that expresses that every reachable state fulfils Q. In addition, a state transition assertion may be given by the construct

**invariant** R(a, a')

restricting the state changes to those which lead from state a to state a' with $R(a, a')$. Finally, a state assertion

**init** P(a)

may be given defining the initial properties. Every invariant $Q(a)$ restricts also the set of initial states.

An invariant R for an export interface expresses that each method call fulfils R. This means that $R(a, a')$ holds for every invocation where a is the attribute value of the state before and a' is the attribute values of the state after the invocation.

**Example.** State transition invariant
For the attribute $a$ the relation $a' = a + 1$ used as an invariant expresses that each method invocation increases the value of a by one. □

There are two essentially different ways to interpret invariants in specifications by contract for classes. One way is to require that each method respects the invariant. The other sees in the invariant an additional specification ("constraint") for each method. We choose the second alternative.

In our case we add each invariant given by the condition Inv and for every method to its precondition P and to its post-condition Q. This way we get for the modified precondition

$$P(a, w, v) \wedge Inv(a)$$

and the modified post-condition where we add each transition invariant

$$Q(a, w, v, a', v') \wedge R(a, a') \wedge Inv(a')$$

In the reminder of this paper, invariants are seen as implicit parts of the initial state assertions, all preconditions and all post-conditions. This saves some notational overhead.

For the class List it is rather straightforward to provide a specification by contract based on the algebraic specification of sequences.

**Example.** List interfaces
An interface for the class List is given by the following set of messages. We use the attribute v : **Var** Seq Data. The specification by contract for the interface List reads as follows:

**interface List** {

v : **Var** Seq Data;
**initial** v = $\langle\rangle$;

**method** add (d : Data);
**pre** true
**post** $v' = v^\circ\langle d\rangle$

**method** size (Var s : Nat);

**pre** true
**post** $s' = length(v) \wedge v' = v$

**method** get (i : Nat, r : **Var** Data);
**pre** $1 \leq i \leq length(v)$
**post** $r' = ith(i, v) \wedge v' = v$

**method** contains (d : Data, r : **Var** Bool);
**pre** true
**post** $r' = (index(d, v) > 0) \wedge v' = v$

**method** indexOf (d : Data, r : **Var** Nat);
**pre** $index(d, v) > 0$
**post** $r' = index(d, v) \wedge v' = v$

**method** remove (d : Data);
**pre** $index(d, v) > 0$
**post** v' = drop(d, v)

}
Invariants restrict the set of reachable states. In this example we do not have to formulate an invariant, since every state in the state space can be reached. However, we can easily switch to a version with a specification that shows an interesting invariant. Assume we introduce a further attribute

le : **Var** Nat

into the class List which stores the length of the sequence represented by attribute v. Then obviously we always assume the following assertion to hold

**invariant** le = length(v)

We may add this equation as an invariant to the class. By our interpretation of invariants this implies for our contracts for the methods that every method that changes v changes the attribute le accordingly. □

A specification by contract treats an export interface as a state transition system. Thus the specification defines, essentially, a state machine by restricting the state space and the state changes by the specification by contract.

**Example.** Cell
We give another simple example of a class defining a storage cell.

**Class** Cell =
{ c: **Var** Data | {void}
**initial** c = void

**method** store (d: Data)
**pre** c = void
**post** c' = d

**method** read (v: **Var** Data)
**pre** $c \neq void$
**post** $c' = c \wedge v' = c$

**method** delete ()
**pre** $c \neq$ void
**post** c' = void

}
This defines the interface of a simple memory cell. Again there is no nontrivial invariant involved since all states are reachable. □

One way of looking at the objects of a pure export class is to see them as state machines. This view is explained in more detail in the following section.

*3.2.3. Export Interfaces described by State Machines*

The specification by contract takes an atomic state transition view. Every method invocation results in an atomic state transition. The pre- and post-conditions characterize the states under which such an invocation can take place to guarantee a certain property of the generated state. In this section we show that this way essentially a state machine is defined (see also [15]).

**Definition.** Class state machine for an export interface
Given an export interface with an attribute set and a set of methods M the associated state transition function is a partial function of the form

$$\Delta : \Sigma(V) \times \text{INVOC}(M) \to (\Sigma(V) \cup \{\perp\})$$

Here for $m \in \text{INVOC}(M)$ and $s, s' \in \Sigma(V)$ the equation $\Delta(s, m) = s'$ expresses that in state $s$ the method invocation $m$ is enabled and leads to the state $s'$ (note that $m$ includes the results of the invocation – thus if $m$ is not enabled in state s it may simply mean that the results indicated in $m$ of the method invocation cannot occur). If $\Delta(s, m)$ does not have a defined result, this means that the method invocation $m$ is not enabled in state $s$. $\Delta(s, m) = \perp$ expresses that the method invocation does not terminate. In addition, we assume a set of initial states $I\Sigma \subseteq \Sigma(V)$. □

In the definition above we have defined deterministic state machines. In general, we have to deal with nondeterministic state machines of the form:

$$\Delta : \Sigma(V) \times \text{INVOC}(M) \to \wp(\Sigma(V) \cup \{\perp\})$$

The state machine associated with a class is easily defined via its specification by contract.

Given a method invocation $m(c_1, c_2, w, v, v')$ for method m with precondition $P(a, w, v)$ and post-condition $Q(a, w, v, a', v')$, we get (if the call terminates) a specification of a state transition function as follows:

$$\Delta(\sigma, m) = \{\sigma' : P(\sigma(a), w, v) \wedge Q(\sigma(a), w, v, \sigma'(a), v')\}$$

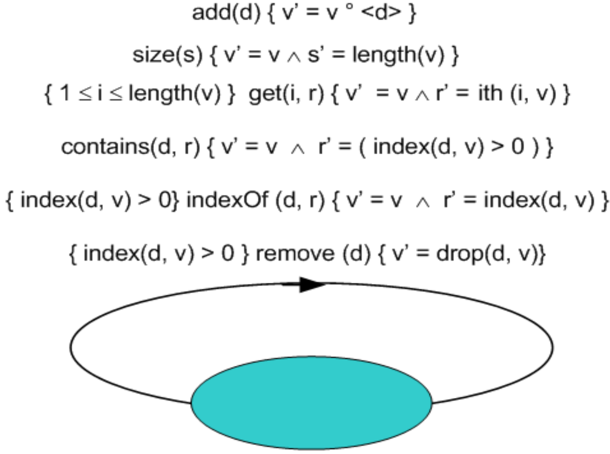



**Figure 1.** State transition diagram that describes the state machine of a List

By $\sigma(a)$ we denote the value of the attribute a in state $\sigma(a)$. The state transition diagram in Fig. 1 shows only a slightly different way to represent the information by a specification by contract. It is a simple version of a state transition diagram with only one control state. Since the set $\Sigma(V)$ is in general infinite, it is quite common in practice to pick a finite abstraction of $\Sigma(V)$, implying a description of $\Delta$ as presented in Fig. 1.

Perhaps more appropriate than the diagram is in this case a table as shown below.

| Precondition | Method Invocation | Post-condition |
|---|---|---|
| | add(d) | $v' = v^\circ\langle d\rangle$ |
| | size(s) | $v' = v \wedge s' = length(v)$ |
| $1 \le i \le length(v)$ | get(i, r) | $v' = v \wedge r' = ith(i, v)$ |
| | contains(d, r) | $v' = v \wedge r' = (index(d, v) > 0)$ |
| index(d, v) > 0 | indexOf(d, r) | $v' = v \wedge r' = index(d, v)$ |
| index(d, v) > 0 | remove(d) | $v' = drop(d, v)$ |

Some heuristic for picking the abstraction may be to group equal transitions or to split complex transition relations into smaller ones on different states.

The difference between a state transition diagram specification of a class and a specification by contract is mainly a methodological one. In the first case we consider the states and define which method calls are possible in each state and to which successor state they lead. In the second case we specify for each method in which states they may be invoked leading to which successor states. Formally this can be seen just as two ways of specifying the state transition relation: For each state $\sigma \in \Sigma(v)$ we define

$$\Delta_\sigma\text{:}(\text{INVOC}(M) \rightarrow \wp(\Sigma(V)))$$

For each method $m \in \text{INVOC}(M)$ we define

$$\Delta_m\text{:}(\Sigma(V) \rightarrow \wp(\Sigma(V))$$

We get for $m \in M$ and $c \in \text{INVOC}(m)$:

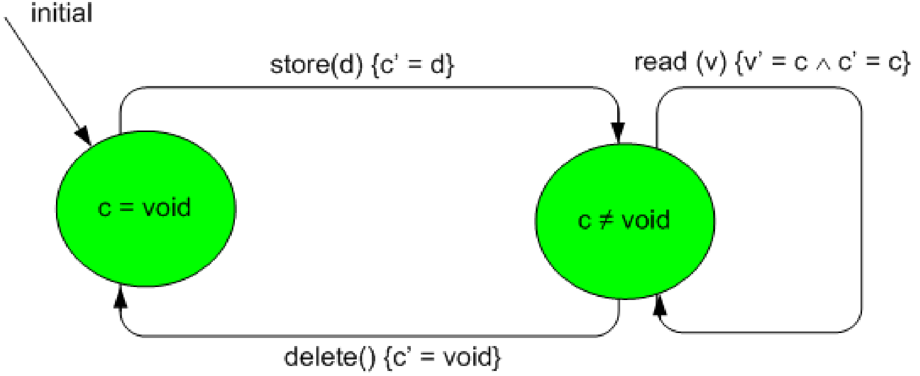



**Figure 2.** State Transition Diagram for the Interface of the Cell

$$\Delta_\sigma(c) = \Delta(\sigma, c)$$

and

$$\Delta_m(\sigma) = \Delta(\sigma, c)$$

We also may use tables as already shown above to define the behaviour of classes as well as state transition diagrams, which actually are just a graphical representation of the tables.

**Example.** Memory Cell
The memory cell is specified by contract easily as shown in the previous section. This defines the interface of a simple memory cell. Here there is no nontrivial invariant involved since all states are reachable. It is easy to provide a state transition description for the state machine modelling a cell as it is shown in Fig. 2. □

Export interfaces are simple to specify, since they can be modelled by atomic state changes.

### *3.2.4. Reachable States*

For a state machine given by a set of initial states $\Sigma \subseteq \Sigma(V)$ and a state transition function

$$\triangle : \Sigma(V) \times \text{INVOC}(M) \rightarrow (\Sigma(V) \cup \{\perp\})$$

we define the set of reachable states $\Sigma_R \subseteq \Sigma\ (V)$ as the least set that fulfils the following formulas:

$$\Sigma_0 \subseteq \Sigma_R$$

$$\sigma \in \Sigma_R \Rightarrow \triangle(\sigma, c) \in \Sigma_R$$

Every predicate

$$p: \Sigma\,(V) \,\rightarrow \mathbb{B}$$

with

$$p\,(\sigma) \;\Rightarrow \sigma \in \Sigma_R$$

is called an *invariant*. Every predicate with

$$p\,(\sigma) \Rightarrow \; p\,(\triangle\,(\sigma, c))$$

is called *stable*. Note that not every invariant is stable and not every stable predicate is an invariant. If for a stable predicate p we have

$$\sigma \in \Sigma_0 \Rightarrow \; p\,(\sigma)$$

then p is an invariant. This is the key idea how to prove that a predicate q is an invariant: find a stable predicate p that implies q and holds for the initial states. Since every export interface specification defines a state machine the concept of stable and invariant predicates carries immediately over to classes.

### *3.2.5. Closed View of Export Interfaces: Systems*

By classes with export interfaces we get a closed view onto object-oriented systems. Two systems with export-only interfaces cannot be composed in a nontrivial way since all we can do with these systems is to call their methods. The systems never "call back" but simply respond by a return message. We speak of closed systems. Therefore we may conclude that such classes describe systems, but not general components.

For such closed system we get a simple concept of observability. What we can observe is the sequences of method calls, and, in particular, whether method calls terminate and which results they produce.

## **4. Limitations of Export Interfaces and Specification by Contract**

So far our view onto class interfaces is simple. Every method invocation corresponds to one atomic state change. This simplicity goes away if we are interested modeling forwarded method calls explicitly.

### *4.1. Forwarded Method Invocations*

To be able to compose two components in a way that they cooperate they have to exchange information. The only sensible way to do this in conventional object orientation is by mutual method invocation. The possibilities to allow for such forwarded calls and to compose components on this basis are discussed in the following.

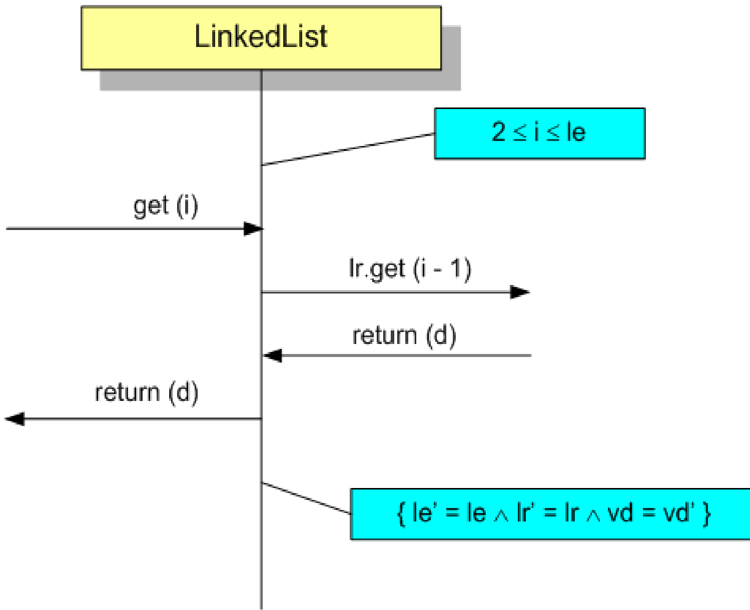



**Figure 3.** Message sequence chart for the method get

The specification of export-only interfaces is particularly simple since it can rely on a simple control flow. By each method invocation exactly one state transition is executed. The control is transferred to the component and returned at once. A method invocation is understood as an atomic action that corresponds to a possible huge state change this way. This is also called the synchronous view for method invocations. This makes the execution model extremely simple - too simple for the component world.

The simplicity of this situation changes significantly if we allow and consider additional invocations of methods (especially of methods of other classes that are not in the export interface) during the execution of methods. We speak of forwarded method calls. This way we get a considerably more complex execution model. A method invocation can be seen as an atomic state change then, only if we comprise also state changes for the objects affected by the forwarded method invocations in this state change. As a consequence the method invocations change not only the local attributes of the called object, but also those of other objects, in the general case also of objects that do not belong to classes in the considered sub-system.

In this section we consider the semantic consequences of further invocations of methods during the execution of method calls. We study families of classes (and their objects) that encapsulated states of which are changed by a method invocation by forwarded method calls.

### 4.2. Interactive Method Invocation Illustrated by MSCs

In this section we do not understand method calls as events that result in atomic huge state changes for all the objects affected by forwarded message calls, but consider the addressed class and object in isolation. A message sequence chart is a good way of representing an instance of the interaction behaviour of an interactive method invocation

and illustrates forwarded method invocation. An example for a method get is shown in Fig. 3.

Here we simplify the representation of the method invocation messages and the return messages in diagrams. We do not list the identifier of the object in the invocations explicitly. The message return(d) stands for a return message with the return variable r with r = d. We can give or even generate a message sequence chart for each of the state transitions, however, the number of message sequence charts can be very high - even infinite. We study the situation of forwarded method invocations in a more systematic way in the following section.

### *4.3. A Motivating Example for Forwarded Method Invocations*

In this section we study a more involved example of a list implementation where lists are realized by a family of pointer structures. This example serves us to discuss problems of forwarded method calls.

**Example.** Lists interfaces for pointer classes
A syntactic interface of the class List is given by the class LinkedList and the following set of methods. Here we do no longer use the attribute

v : **Var** Seq Data

but each object encapsulates only one element of the sequence as well as links to the objects carrying the other elements.

The implementation - now given for illustration purposes by explicit program code and not by a specification - reads as follows:

```
Class LinkedList

{       vd:   Var Data
        lr:   Var LinkedList
        le:   Var Nat
        initial: le = 0 ∧ vd = Nil ∧ lr = Nil
        invariant: le > 0 ⇔ lr ≠ Nil

  method add (d : Data):
        if le = 0 then vd := d; le := 1; create.LinkedList(lr)
                  else le := le+1; lr.add(d) fi

  method size (s : Var Nat):
        s := le

  method get (i : Nat, r : Var Data):
        if i = 1 then r := vd else lr.get(i-1, r) fi

  method contains (d : Data, r : Var Bool):
        if le = 0 then r := false else if vd = d then r := true
                                                else lr.contains(d, r) fi fi
```

```
method indexOf (x : Data, r : Var Nat):
    if vd = d then r := 1 else lr.indexOf(d, r); r := r+1 fi

method remove (d : Data):
    if vd = d then if le > 1then lr.get(1, vd); lr.remove(vd), le := le-1
                            else le := 0; vd := Nil
                   fi
              else le := le-1; lr.remove(d)
    fi
}
```

However, this example is not elegant. To improve it we could rather choose perhaps a different set of methods, perhaps private methods, to provide a more elegant version of this class. We are not interested in such more elegant versions here, however. Moreover, the method remove is - as it is implemented - not efficient, since, when invoked, it goes down the complete list. A more efficient version is obtained as follows:

```
method remove (d : Data):
    if vd = d then if le > 1 then lr.get(1, vd); lr.assignrest(lr), le := le-1
                             else le := 0; lr := Nil
                   fi
              else le := le-1; lr.remove(d)
    fi

method assignrest (r : Var LinkedList):
    r := lr
```

This change leads, however, a step deeper into programming with pointers. We are interested in the principle problems of specifying the behaviour of interfaces with forwarded method invocations and not in pointer structures, which pose additional problems. A difficult issue, in fact, is to give a specification by contract for that class according to the forwarded method invocations. Actually, to give an appropriate comprehensive invariant is already difficult. Actually the invariant should express that the links form a chain of elements of length le without cycles. This can only be expressed informally by introducing sophisticated operators as shown below(see [17]). □

The class LinkedList is compatible to the class List and vice versa. Therefore they provide the same interfaces. However, this is not captured at all in the specification by contract. Actually the class LinkedList fulfils the specification by contract for the class List as shown above, however, now the state consists not just of the attributes of the respective object that represents a sequence, but the sequence is represented by a linked list of several objects.

To deal with this problem we see two options. Either we try to express the contract specifications at the level of the object data structures. Then we need a notation that allows us to express properties of the linked structure of the object data model in terms of the attributes of referenced objects (see [17]). Another technique is to abstract from that concrete object state into a sequence and then talk about the sequence in the contracts as before. This way we collect several objects into one state.

**Example.** Specification by contract for the pointer class List
Given an object b of type LinkedList we can calculate the sequence represented by b by the following function:

```
Fct absseq (b : LinkedList) Seq Data:
    if b.le = 0 then ⟨⟩ else ⟨b.vd⟩ ∘ absseq(b.lr) fi
```

Here we use the notation b.a to refer to attribute a in the of the object b. This function calculates for every object of type LinkedList, which is not cyclic, a sequence that is represented by the linked list. If the LinkedList is cyclic the computation does not terminate and the value of the function call is not defined.

Given this function we easily write a specification by contract. The specification by contract reads as shown below. Here we write self to refer to the respective object identifier:

```
Class LinkedList1 {
    vd : Var Data
    lr : Var List
    le : Var Nat
    initial : le = 0 ∧ lr = 0
    invariant: le > 0 => lr ≠ Nil

  method add (d : Data):
    pre true
    post absseq(self') = absseq(self) ∘ ⟨ d ⟩

  method size (Var s : Nat):
    pre true
    post s' = length(absseq(self)) ∧ absseq(self') = absseq(self)

  method get (i : Nat, r : Var Data):
    pre 1 ≥ i ≥ length(absseq(self))
    post r' = ith(i, absseq(self)) ∧ absseq(self') = absseq(self)

  method contains (d : Data, r : Var Bool): pre true
    post r' = (index(d, absseq(self)) > 0) ∧ absseq(self') = absseq(self)

  method indexOf (d : Data, r : Var Nat);
    pre index(d, absseq(self)) > 0
    post r' = index(d, absseq(self)) ∧ absseq(self') = absseq(self)

  method remove (d : Data);
    pre index(d, absseq(self)) > 0
    post absseq(self') = drop(d, absseq(self))
}
```

In fact, this example specification looks again not elegant. Our notation is not clean. In fact, it is not "referentially transparent". Actually, the value denoted by self is not

changed in any of the methods. Therefore writing self' does not make sense. We use this notation, however, to express that we calculate the sequence absseq(self') in the state after the method invocation. □

This example provides a specification of lists, too. In fact, the observable interface behaviour of List and LinkedList are identical although the specifications by contract look quite differently. The proof that both classes have the same interface based on the specification by contract techniques needs a sophisticated theory.

## 5. Open View: Components with Export and Import

In this section we develop a concept of a component for object orientation. A component is a syntactic unit that can be composed.

### *5.1. Methods, Invocations and Return Messages*

In this section we introduce an approach to interfaces and components for classes and forwarded method invocations. In specification by contract we treat method invocations as atomic state changes. Now we treat method invocations as sequences of state changes, starting with the method invocation message and ending with the corresponding method return message. In the latter case, the asynchronous case, method invocations correspond to two messages.

**Definition.** In- and Out-Messages for a method header
A method invocation consists of two interactions of messages called the method invocation message and the return message. Given a method header (for explanations see above)

**method** m (w : WT, v : Var VT)

the corresponding set of invocation messages is defined by the following equation

SINVOC(m) = m(b1, b2, w, v): w ∈ WT, v ∈ VT, b1, b2 ∈ Object

Here we treat variables as call-by-value-return parameters. The v represents the value of the variable parameter before the call. The return message has the type (where v' is the value of the variable after the execution of the method invocation)

RINVOC(m) = return_m(b1, b2, v'): v' ∈ VT, b1, b2 ∈ Object

With each method we associate this way two types of messages, the invocation message and the return message. □

Given a set of methods M we define the sets of invocation and return messages as follows:

$$\text{SINVOC}(M) = \bigcup_{m \in M} \text{SINVOC}(m)$$

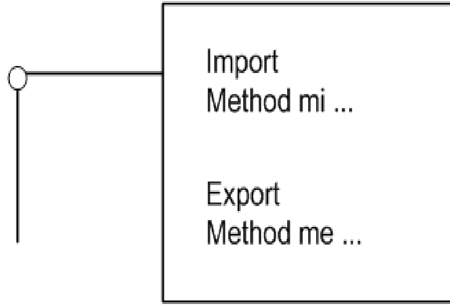



**Figure 4.** Graphical representation of an export/import interface

$$\text{RINVOC}(M) = \bigcup_{m \in M} \text{RINVOC}(m)$$

This way we denote the set of all possible invocations of methods that are in the set of methods M.

### *5.2. Export/Import Interfaces*

In object orientation, a class uses other classes via their methods as sub-services to offer its interface behaviour. Thus a class on one hand offers methods to its environment called an *export* method and on the other hand invokes methods of other classes called *import* methods. We speak of the *exported* methods and the *imported* methods of a class. This idea of exported and imported methods is captured by import/export interfaces.

In the following we deal with issues related to the import and export of interfaces in more detail.

#### *5.2.1. Syntax of Export/Import Interfaces*

In the case of forwarded calls we deal with classes dealing with two kinds of methods, imported and exported ones. This should be explicitly reflected in the syntactic and semantic interface of object oriented systems. Every interface specification with an explicit import part and an explicit export part defines a so-called export/import interface.

**Definition.** Syntactic export/import interface
A *syntactic export/import interface* consists of two syntactic "export" interfaces represented by two sets of class names, sets of method headers associated with each class name, which define the set of export and the set of import methods. Methods in the set of export methods can be called from the environment, import methods are methods provided by the environment and can be called by the component. □

For simplicity, we assume that all methods and classes in the export and import interfaces have different names, since we do not want to deal with overloading. Moreover, we do not treat explicitly in the following the exported types (class names). Given an export/import interface of a component c we denote by EX(c) its export interface and by IM(c) its import interfaces, both being simple export interfaces. A syntactic export/import interface can easily be described graphically as it is shown in Fig. 4.

A class in object oriented programming, in general, delegates parts of the execution to other classes via forwarded method invocations and therefore in general has an export/import interface in spite of the fact that the idea of explicit imported interfaces is surprisingly not supported by most of the conventional object oriented techniques. Often the import interface is kept implicit for classes and not mentioned at all in the syntactic interface description.

Now we give a first example of an export/import interface and its specification by a state machine.

**Example.** Accountmanager
We consider a simple class Accountmanager that is an account manager. It is based on the following three types:

| | |
|---|---|
| Person | the type of individuals that may own accounts |
| Account | the type of accounts (a class) |
| Amount | the type of numbers representing amounts of money |

For the class Accountmanager we consider only one export method and one import method. It uses a function f

**Fct** f = (x: Person) Account

that relates persons to their account numbers.

**Class** Accountmanager =
{...
**export method** credit = (x: Person, y: **Var** Amount, z: **Var** Account)
...
**import method** balance = (y: **Var** Amount)
}

The account manager calls the method balance of object b, which is a manager of the money in the account. But for finding out the credit for other persons the credit has to be determined by issuing back calls. Therefore the proper state machine looks as shown in Fig. 5 which gives such a state machine with input and output for the account manager. In this diagram we write on the arcs, which represent state transitions,

{A} m1 / m2 {B}

to express that this transition is executed if the assertion A holds and the message m1 is received (as input); as an effect of this transition the message m2 is send (produced as output) and the state change described by the state transition assertion B takes place.

Note that the state machine requires additional attributes that are not the attributes that we use in the class Accountmanager such as

b: **Var** Object
p: **Var** Person

to store the actual parameters of the call while waiting for the result of the forwarded call. They can be seen as examples of simple representations of the call stack. □

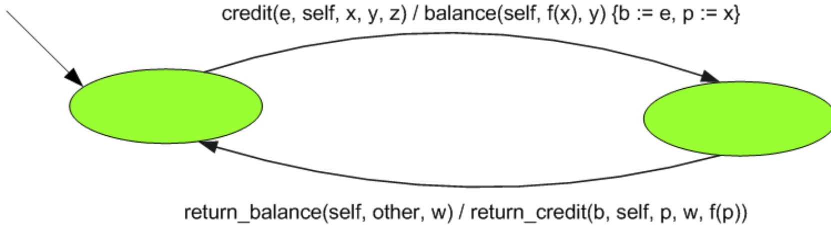



**Figure 5.** Graphical representation of an export/import interface

The example shows also a property more explicitly that is implicit in classes. An object cannot be revisited by method calls (in this case of method invocations of message credit) before the call is finished (if we assume that objects are like monitors). If we do not like this rule, we have to provide another state machine. This will be discussed in more detail below.

The example also shows that there are cases where the call stack is part of the state space of the state machine associated with a class. At the level of object-oriented code this call stack is implicit. When writing specifications, we have to make the call stack explicit.

### 5.2.2. *Control Flow, Forwarded Calls, Back-Calls, and the Call Stack*

When dealing with export/import interfaces we have to deal with call-backs, in general. In other words, a method invocation for object b may lead to a forwarded call that in turn may lead to invocation of methods of object b. We speak of a *call-back*. For forwarded calls and possible call-backs, we need additional state attributes in the local state of the interface to be able to find the correct continuation for returned invocations (in the sense of the return addresses for subroutine calls and for storing the parameters).

In the general case, we have to work with a full call stack. The call stack is the classical way to manage nested procedure calls or method invocations. Every time a procedure or method is called, the parameters and the return address are pushed onto the call stack. This way the call stack has to deal with control (such as return addresses) as well as data aspects (such as parameter values) of pending calls. The call stack determines the continuation after the return of a forwarded call and provides the local state information providing the values of the parameters of the call under execution.

### 5.2.3. *Modelling Export/Import Interfaces by I/O State Machines*

In this section we demonstrate how to describe the behaviour of export/import interfaces by state machines with input and output. Since we have a set of in- and out-messages related to each of the method headers, this easily generalizes to class interfaces.

**Definition.** In- and Out-Messages of a syntactic class interface
Let c be a syntactic export/import interface with set EX(c) of export class names and their methods and the set IM(c) of import class names and methods. It defines a set In(c) of ingoing messages

$$\text{In(c)} = \text{SINVOC(EX(c))} \cup \text{RINVOC(IM(c))}$$

and of a set of outgoing messages Out(c) specified by

$$Out(c) = SINVOC(IM(c)) \cup RINVOC(EX(c))$$ □

Since we have a set of in- and out-messages related to each of the method headers of an export/import interface, we construct a state machine that describes the behaviour of the export/import interface. It uses the invocation messages in the export interface and the return messages in the import interface as input and the invocation messages in the import interface and the return messages in the export interface as output.

**Definition.** Export/import state machine
Given an interface c with an attribute set V and a set of methods, the associated state machine has the form (here we work with a total function)

$$\triangle : State \times In(c) \rightarrow ((State \times Out(c)) \cup \{\perp\})$$

Here for $m \in In(IF)$ the equation $\triangle(s, m) = \perp$ expresses that the method invocation does not terminate. The state space State is defined by the equation

$$State = \Sigma(V) \times CTS$$

Here CTS is the control state space. Its members can be understood as representations of the control stack. Since we do not want to go deeper into the very technical discussion of control stacks, we do not further specify CTS. Of course, we assume that a set of initial states $IState \subseteq State$ is given. □

A convenient way to describe I/O state machines is a state transition diagram. In the case of asynchronous models of method invocations we work with state machines with input and output called Mealy machines.

It is not difficult to go - in the case of export-only interfaces - from such a given Mealy machine

$$\triangle : State \times In(IF) \rightarrow ((State \times Out(IF)) \cup \{\perp\})$$

to the kind of state machines

$$\triangle': \Sigma(V) \times INVOC(M) \rightarrow (\Sigma(V) \cup \perp)$$

we have introduced for export-only interfaces. In the case of export-only interfaces the only output messages that exist are return messages. Each transition of the state machine

$$(s', y) = \triangle(s, x)$$

determines a transition

$$z' = \triangle(z, c)$$

and vice versa. From

$$(s', y) = \triangle(s, x)$$

we easily construct the data states $z, z' \in \Sigma(V)$ from the states s and s' since in this case the control stack is trivial. The message $c \in INVOC(M)$ with $c = m(b1, b2, w, v, v')$ with $z' = \triangle(z, c)$ is determined by $x = m(b1, b2, w, v)$, $y = m(b1, b2, v')$.

### *5.2.4. Closed View of Export/Import Interfaces: Systems*

By classes with export/import interfaces we can get a closed view onto object oriented systems. This means that we consider a method invocation as one state change that changes the attribute of the called object and all attributes changed by further forwarded method calls.

For the closed view we may again use the specification by contract idea. However, then we have to refer in the assertions of methods to the attributes of other objects that are updated by forwarded calls.

### *5.2.5. Specification by Contract for Export/Import Interfaces*

In the general case of export/import interfaces we have to deal with forwarded calls and back-calls. As demonstrated this enforces to make the call stack explicit part of the state space - also in the case of specification by contract.

In the general case of specification by contract for export/import interfaces assertions have not only to refer to the call stack, but also to attributes of objects in their environment. We demonstrate that again by our example of the account manager.

**Example.** Account manager (continued)
We consider again the simple class Accountmanager. The account manager calls the method balance of an object b, which is a manager of the money in the account. But perhaps this call changes the attributes of object b.

```
Class Accountmanager
{       Fct f = (x : Person) Account:...

        method credit = (x : Person, y : Var Amount, z : Var Account):
            f(x).balance(y); z:= f(x)
}
Class Account
{       a, d : Var Nat; a denotes the state of the account, d what is bound by credit

        invariant a ≥ d;

        method balance = (y : Var Amount)
            if a-d ≥ y then d := d+y
            else if a = d then y := 0
                      else y := a-d; d := a
            fi fi
}
```

In this example a call of the method credit leads to a call of balance, which may change the attribute d. The specification by contract for the method credit reads as follows:

```
        method credit = (x : Person, y : Var Amount, z : Var Account):
                pre    f(x) ≠ nil
                post   z' = f(x)
                ∧      f(x).d' = f(x).d+y'
```

$\wedge$ (f(x).a-f(x).d $\geq$ y $\Rightarrow$ y' = y)
$\wedge$ (f(x).a-f(x).d $\leq$ y $\Rightarrow$ y' = f(x).a-f(x).d)

This shows that we have to refer to attributes of the object f(x) in the method credit. Here we use again the notation b.a to refer to attribute a of the object b. □

In the specification by contract we do not actually refer to the export/import interface, in fact. We do not express that method balance is called to change the attributes of the object f(x). However, we may specify balance in the import specification by contract and then refer to this specification in the assertions for credit:

**Example.** Account manager (continued)
We consider again the simple class Accountmanager. The specification by contract of the export/import interface reads as follows:

**Class** Accountmanager =
{...
**export**
    **method** credit = (x: Person, y: Var Amount, z : Var Account):
        **pre** f(x) $\neq$ nil
        **post** z' = f(x)
        $\wedge$ post(f(x).balance(y))
...
**import**
    **method** balance = (y: **Var** Amount):
        **pre** true
        **post** d' = d+y'
        $\wedge$ (a-d $\geq$ y $\Rightarrow$ y' = y)
        $\wedge$ (a-d $\leq$ y $\Rightarrow$ y' = a-d)
}

Here post(b.m(y)) stands for the post-condition of the method m modified by replacing all local attribute identifiers such as a by the global identifiers b.a. □

The example looks fine but it does not show an additional difficulty: Often several forwarded calls are executed in specific local states. The order in which the forwarded methods are called and in which local states introduces another difficulty here. This problem can be solved, but makes the specification more execution specific and much more incomprehensible.

### *5.2.6. Observability for Export/Import Interfaces*

For an interface with export and import it makes an essential difference how a system is seen from the import/export point of view either making the import explicit or keeping it implicit. For a useful interface description for architectures, we have to make the import explicit. This leads to different idea of observability. Now we observe sequences of alternating input and output actions as well as the termination of method invocations.

Actually we have now two ways of non-termination. In one case an input message m in a state s may not lead to an output message. This is indicated by $\triangle(s, m) = \perp$. Moreover, a method invocation will lead to an infinite sequence of in- and out-messages under certain reactions of the environment.

In this model of observability using I/O state machines we can even do a step in the direction of concurrency using interleaving. Assume, we send a method invocation message to a component that triggers an invocation of an import method representing a forwarded method call. Then a sequential execution the next input message could only be another invocation message (triggered by a back-call) or the return message to the previous call. But nothing prevents us from giving an arbitrary invocation message (which cannot be distinguished from a back-call) and thus to handle interleaved independent method invocations. We only have to place return messages at the right places in the input streams (see later). Thus we get a restricted form of concurrency.

#### *5.2.7. Concurrency and Multi-Threading*

So far we have mainly considered sequential control flow without concurrency. In more technical terms we did only consider executions of one thread. This kept our execution model rather simple. In large distributed systems a more complex model is mandatory. There are several threads executed concurrently. Then it is no longer valid that a method invocation leads to a sequence of method invocations and return messages that is completed before the next method invocation takes place. New method invocations from other threads may arrive before a method invocation sequence is completed. Several method invocations are executed, in general, in an interleaving mode.

Decomposing a method call into two complementary message exchanges, the invocation call and the return, which is done to be able to have open specifications of components and classes gives an interesting additional option: now we may (or may not) accept further method invocations before a method call has been completely executed - before it has sent back its return message. This forces us to freely introduce interleavings of calls and to introduce language constructs that allow us to avoid them in cases where calls should be completed before further calls are processed (mutual exclusion).

As a result of concurrency and multi-threading we get interleaving of single threaded invocation sequences. This leads also to issues of synchronization to be able to control the interleaving. Note that now the invocation stack has to be replaced by an individual stack for each thread.

#### *5.2.8. Control Flow in Export/Import Interfaces*

The specification of export/import interfaces gets more involved than that of simple interfaces since in this case we can rely no longer on the simple control flow of method invocations in terms of atomic state changes. By each method invocation a sequence of state transitions is executed alternating between those invocations changing the local state of the considered interface and those triggering state changes in the environment and producing return messages or even further invocations of messages for the considered interface (so called *call backs*).

The control is then transferred several times back to the environment and returned from the environment several times while executing a method invocation. This control flow transfer corresponds to a sequence of method invocations and method return messages. This sequence has a specific structure as shown by the following BNF form:

SeqMIR ::= EMI { IMI {SeqMIR}* IMR }* EMR

Here EMI denotes the set of invocations of export methods, IMI denotes the set of invocations of import methods, EMR denotes the set of return messages of export methods and IMR denotes the set of return messages of export methods. SeqMIR denotes the set of sequences of invocation and return messages that may occur in principle (syntactically) as results of method invocation of export methods.

Due to application specific logical constraints the set of actually occurring sequences of method invocation messages and return messages is a subset of SeqMIR. This subset specifies the invocation protocol. The formula above simply says that sequences of invocation and return messages always start with an invocation of an export method and always ends with the corresponding return message. If there is a sequence between these two messages, this sequence starts with an invocation of an import method and always ends with the corresponding return message. The sequence between these messages can again be a sequence of invocation sequences for export methods.

Since invocation sequences always have this regularity, each return message can be related uniquely to its corresponding invocation message and vice versa. Note, the grammar describing the language SeqMIR and the language itself is a Chomsky-2-language (context-free language) and thus needs a stack to parse it. This corresponds to the call stack existing for each thread at run time.

### *5.3. Relating Export/Import Interfaces: Design and Refinement*

As we have demonstrated, for object oriented interfaces in its most general form with call forwarding and call-backs we have to make the call stacks explicit and refer to the local attributes of the environment in the assertions when working with specification by contract. This makes specification by contract more difficult and less modular. In the following we introduce a refinement relation for classes and objects in terms of specification by contract. It provides an answer to our principle of substitutability and compatibility.

#### *5.3.1. Design By Contract*

In this section we briefly outline the key idea of design by contract. We explain how to connect specification by contract to implementations. We deal with export/import interfaces.

We explain the general idea of an implementation in terms of design by contract based on a specification by contract for an export/import interface specification. Given specifications by contract for both the import and the export interface (we assume for simplicity that all invariants are included explicitly in the pre- and post-conditions) we construct a "verified" implementation as follows. We give code for every method in the export part and prove that the code fulfils the pre- and post-condition specification as given by the contracts. In the proof we may use for all forwarded calls the pre- and post-condition from their design by contract assertions. The proof can be, for instance, done by annotating the code using Hoare logic (see [8]).

**Example.** Account manager (continued) We consider again the simple class Account-manager. The account manager calls the method balance of an object b, which is a manager of the money in the account. We do not give the proof.

```
Class Accountmanager
{       Fct f = (x : Person) Account: ...
export
        method credit = (x : Person, y : Var Amount, z : Var Account):
                pre     f(x) ≠ nil
                post    z' = f(x)
                ∧       f(x).d' = f(x).d+y'
                ∧       (f(x).a-f(x).d ≥ y ⇒ y' = y)
                ∧       (f(x).a-f(x).d ≤ y ⇒ y' = f(x).a-f(x).d)
                body    f(x).balance(y); z:= f(x)
  }
  import

                a, d : Var Nat;

                invariant a ≥ d;

                method balance = (y : Var Amount):
                        pre     true
                        post    d' = d+y'
                        ∧       (a-d ≥ y ⇒ y' = y)
                        ∧       (a-d ≤ y ⇒ y' = a-d)
  }
```

The proof that the body of the method credit is correct with respect to the pre/post-condition is quite straightforward using the pre/post-condition of the method balance.

□

A design by contract including a proof for a component with export and import methods reads as follows:

Step 1: Specify: Specification by contract (SbC): We give SbCs for all methods
Step 2: Design: Component implementation

- We provide a body for each exported method
- Only method calls are allowed in the bodys that are either in the export or import parts (no calls of "undeclared" methods)
- The body is required to fulfil the pre/postconditions

Step 3: Verify: Component verification

- Verify the pre/post-conditions for each implementation of an export method
- We refer to the SbCs for the imported (and the exported) methods use in nested calls in the bodies when proving the correctness of each exported method w.r.t. its pre/postconditon

An interface specification is called *correctly implemented* if for every export method a body ("code") is given with an assertion proof along the lines described above.

There is some similarity to Lamport's TLA (see [10]) where systems are modelled by

- The set of actions a system can do
- The set of actions the environment can do
- Actions are represented by relations on states
- Fairness/lifeness properties by temporal logic on system runs
- Difference: actions are atomic - method calls are not

We may, in addition, structure the export and import part into (see chapter 6)

- a set of pairs of export and import signatures that are sub-signatures of the overall export and import interfaces
- This pairs may be called sub-interfaces
- This leads in the direction of connectors

Given export/import components $c_i$ with i = 1, 2, and export signature EX($c_i$) and import signature IM($c_i$) we assume that $\mathfrak{R}(\{c_1, c_2\})$ holds, if there are no name conflicts. Then export signature EX and import IM of the result of the composition $c_1 \otimes c_2$ is defined by

$$\text{EX}(c_1 \otimes c_2) = (\text{EX}(c_1)\backslash\text{IM}(c_2)) \cup (\text{EX}(c_2)\backslash\text{IM}(c_1))$$
$$\text{IM}(c_1 \otimes c_2) = (\text{IM}(c_1)\backslash\text{EX}(c_2)) \cup (\text{IM}(c_2)\backslash\text{EX}(c_1))$$

The composed component c = $c_1 \otimes c_2$

- exports what is exported by one of the components and not imported by the other one and
- imports what is imported by one of the component and not exported by the other one.
- Methods that imported by one component and exported by the other one are bound this way and made local

Actually we get local (hidden) methods that way, we ignore that to keep notation simple.

*5.3.2. Verification of composed components*

Let all definitions be as before and assume given SbC for all methods. For proving the correctness of composition we prove

- for each exported method m with pre-condition $P_{ex}$ and post-condition $Q_{ex}$
- that is bound by some imported method m with pre-condition $P_{im}$ and post-condition $Q_{im}$ such that

$$P_{im} \Rightarrow P_{ex} \qquad\qquad Q_{ex} \Rightarrow Q_{im}$$

This gives the general pattern for the d Design by contract for the export/import case:

Step S: Specify system: Export only SbC
Step A: Develop the architecture
 Step AD: Design architecture: List components and their export/import methods
 Step AS: Specify architecture: Give Export/Import SbC for all components
 Step AV: Verify architecture: Show that for the exported method calls the specification be contracts can be proved.
Step I: Component implementation
 Step ID: Design: We provide a body for each exported method

Only calls are allowed that are either in the export or import parts
(no calls of "undeclared" methods)
Step IS: Specification taken from architecture: The body is supposed to fulfil the pre/post-conditions
Step IV: Component verification: SbCs for imported methods are used when proving the correctness of each exported method for its pre/postconditon
Step G: Component composition - integration: correctness for free

This gives a general scheme how to proceed in design by contract and to keep component implementation and verification and architecture verification independent.

*5.3.3. Refinement*

In this section we discuss a notion of refinement that fulfils the idea of substitutability.

Given two interfaces described by specifications by contract, we relate them by relating their states and their pre- and post-condition.

**Definition.** Refinement
Let two export/import interfaces IF1 and IF2 be given, specified by contract. Let V1 and V2 be their attribute sets. We call IF2 a *refinement* of IF1 if the following conditions hold:

- EX(IF1) $\subseteq$ EX(IF2) and IM(IF2) $\subseteq$ IM(IF1); this means that the refined interface offers more methods and uses less.
- There exists a function $\varrho$: $\Sigma$(V2) $\rightarrow$ $\Sigma$(V1) called *state mapping* such that
  - For the specified invariants I1 and I2 (and in analogy for the initial state assertions) of the interfaces we require for all $\sigma 1 \in \Sigma(\text{V1})$, $\sigma 2 \in \Sigma(\text{V2})$:

$$I1(\sigma 1) \Rightarrow \exists \sigma 2 \in \Sigma(V2): \rho(\sigma 2) = \sigma 1 \wedge I2(\sigma 2)$$

$$I2(\sigma 2) \Rightarrow I1(\rho(\sigma 2))$$

  - For all methods m in EX(IF1) (let P1 and Q1 be the pre- and post-conditions of m in IF1 and P2 and Q2 be the pre- and post conditions of m in IF2) we require:

$$P1(\rho(\sigma 2)) \wedge I2(\sigma 2) \Rightarrow P2(\sigma 2)$$

$$Q2(\sigma 2) \wedge I2(\sigma 2) \Rightarrow Q1(\rho(\sigma 2))$$

  - For all methods m in IM(IF2) (let P1 and Q1 be the pre- and post-conditions of m in IF1 and P2 and Q2 be the pre- and post conditions of m in IF2) we require:

$$I2(\sigma 2) \wedge P2(\sigma 2) \Rightarrow P1(\rho(\sigma 2))$$

$$Q1(\rho(\sigma 2)) \wedge I2(\sigma 2) \Rightarrow Q2(\sigma 2)$$

□

These formulas mimic the situation where we replace an implementation with interface IF1 by an implementation with interface IF2. Doing so, we assume that in the implementation of IF2 (and also in that of IF1) we call imported methods. Every time we call an imported method in the implementation of IF2 we assume when calling it that the pre- and post-conditions according to the import specifications hold. This is done in state, say $\sigma 2$. Seen in terms of the original program, this is as if the method is called in state $\rho(\sigma 2)$. The refinement condition then guarantees that also the original precondition $P1(\rho(\sigma 2))$ holds. After the call $Q1(\rho(\sigma 2))$ can be assumed; by the refinement condition we can assume that the assertion $Q2(\sigma 2)$ holds. If we add methods that respect the invariants to the export part of an interface we get a refinement. The definition shows a way to prove refinement relations between interfaces. In this case, since the specification by contract captures all the effects of calls of exported methods, this relation is most relevant. For imported methods the refinement relation is mainly of interest in cases where implementations exist and we want to make sure that these also work in the refined case if we call the imported functions of the interface IF1 instead of those of the interface of IF1.

Refinement is helpful in a number of situations to relate interfaces and components. It supports the stepwise introduction of more and more specific properties.

## 6. Towards a Theory of Components and Architectures in Object Orientation

In this chapter we discuss where we are with our theory of components and architectures in object orientation. In this section we relate the introduced notion of object orientation to those of the theory introduced in section 3. We discuss the state of the art and methodological challenges.

### *6.1.1. What is a Component in Object Orientation*

In object orientation an obvious first choice for the notion of a component is a class. Actually one can argue that rather objects should be considered components. We, however, prefer to see components as building blocks at design time in contrast to objects that are rather building blocks at runtime. So, for our purpose, classes or compounds of classes are an obvious choice. But is a class really a good choice for the notion of a component?

Obviously classes have a lot of properties addressing the idea of components. There is a notion of interface, state encapsulation, and information hiding for classes as we would expect it for components. There are, at least, two arguments, however, throwing some doubts on the idea that classes may be good candidates for components:

- Classes are too small. Actually, of course, one may argue that we can write very large classes. But then we get unstructured huge entities. For components we need larger building blocks with additional hierarchical structuring concepts (the threads of Java are a much too low level concept).
- The concept of concurrency is not support by conventional classes.
- There is no tractable interface specification technique for classes with export and import.

This shows that classes, although they provide concepts close to what we need for components, fail to address necessary requirements for the notion of components.

### 6.1.2. What is Composition in Object Orientation

There is no widely accepted concept of composition in object orientation. Nevertheless, it is not so difficult to define a concept for composition in object orientation. Given two classes with export and import methods (where import methods are related to objects of certain classes), we can compose them in a way, where classes may mutually call methods in their import signature that are in the export signature of the other class. We speak of internal calls. For simplicity, we ignore any problems that may arise with inheritance and method overloading where methods may be called for classes with names that do not occur in the export of the class or methods. So we concentrate on the method names and ignore any aliasing.

We start with the definition, when two classes can be defined. Given classes $c_i$ with i = 1, 2, and export signature EX($c_i$) and import signature IM($c_i$) we define that $\Re$ ({c1, c2}) holds, if there are no name conflicts. Then export signature EX and import IM of the result of the composition c1 ⊗ c2 is defined by

$$EX(c_1 \otimes c_2) = (EX(c_1) \setminus IM(c_2)) \cup (EX(c_2) \setminus IM(c_1))$$

$$IM(c_1 \otimes c_2) = (IM(c_1) \setminus EX(c_2)) \cup (IM(c_2) \setminus EX(c_1))$$

In other words, in the composed class c = $c_1 \otimes c_2$ exports what is exported by one of the classes and not imported by the other one and imports what is imported by one of its component classes and not exported by the other one.

Next we consider the semantic composition of the two state machines associated with the classes or interfaces $c_i$ (i = 1, 2):

$$\triangle_i{:}State_i \times In(c_i) \rightarrow (State_i \times Out(c_i)) \cup \{\perp\}$$

Now we define the composed state machine

$$\triangle{:}State \times In(c) \rightarrow (State \times Out(c)) \cup \{\perp\}$$

as follows

$$State = State_1 \times State_2$$

and for x ∈ In(c) and $(s_1, s_2) \in State_1 \times State_2$ we define:

$$\begin{aligned} x \in In(c_1) \wedge (s_1', y) = \triangle_1(s_1, x) \Rightarrow\ & y \in In(c_2) \Rightarrow \triangle((s_1, s_2), x) = \triangle((s_1', s_2), y) \\ \wedge\ & y \notin In(c_2) \Rightarrow \triangle((s_1, s_2), x) = ((s_1', s_2), y) \\ x \in In(c_1) \wedge \triangle_1(s_1, x) = \perp \Rightarrow\ & \triangle((s_1, s_2), x) = \perp \end{aligned}$$

In other words, we give the input to that state machine to which the input fits. If the output is in the input of the other state machine, we do another state transformation. If this is done forever, then the state transition does not terminate, and thus $\triangle(s_1, s_2)$, x) = ⊥ . In analogy we define the case where the input goes to the second component:

$$x \in In(c_2) \wedge (s_2^{'}, y) = \triangle_2(s_2, x) \Rightarrow y \in In(c_1) \Rightarrow \triangle((s_1, s_2), x) = \triangle((s_1, s_2^{'}), y)$$
$$\wedge \ \ y \notin In(c_2) \Rightarrow \triangle((s_1, s_2), x) = ((s_1, s_2^{'}), y)$$
$$x \in In(c_2) \wedge \triangle_2(s_2, x) = \perp \Rightarrow \triangle((s_1, s_2), x) = \perp$$

This gives a recursive definition for the state transition function $\triangle$ for the composed component. This way we define

$$\triangle = \triangle_1 \parallel \triangle_2$$

Actually, this way of definition results in a classical least fixpoint characterization of the composed transition relation $\triangle$

### *6.1.3. What is a System in Object Orientation*

A system in object orientation in terms of our theory is a class or a set of classes (perhaps a composed one) with an empty import signature. A system nevertheless can actually be composed with a component in an interesting way. Consider a system s with the export set EX(s) and import set IM(s) = ∅ and a component c with EX(s) ⊆ IM(c). Then the term c ⊗ s describes a composed system where s is used as a local sub-system. For two systems composition degrades to the union of the signatures.

### *6.1.4. Intermediate Conclusion*

We have defined a first step of an instance of a theory of components, interfaces, and composition in object orientation. What we presented is certainly not sufficient for practical purposes. However, it gives a first idea what can be achieved and shows the limitations of existing approaches and unsolved problems.

Perhaps, it is worthwhile to draw a bottom line for what we have achieved by our theory and also to draw some conclusions:

- We defined a concept of component in OO as a generalization of the concept of a class: a component is a set of classes and their visible methods, divided into export, import and internal (hidden) ones.
- We described a model for this concept of components, namely state machines with input and output.
- We introduced composition for this concept of components.
- But we pay a (too) high price: we have to make the call stack explicit in the state space of the machine, in general.

There seems to be only one way out: introducing an explicit notion of a component, defining a wrapper for a set of classes and the methods (being the components in object orientation as we have introduced them), and connecting them by asynchronous message passing.

## *6.2. Components in Object Orientation*

Based on the idea of export/import interfaces we define the general notion of a component. Following ideas of architectures where each component is connected over a num-

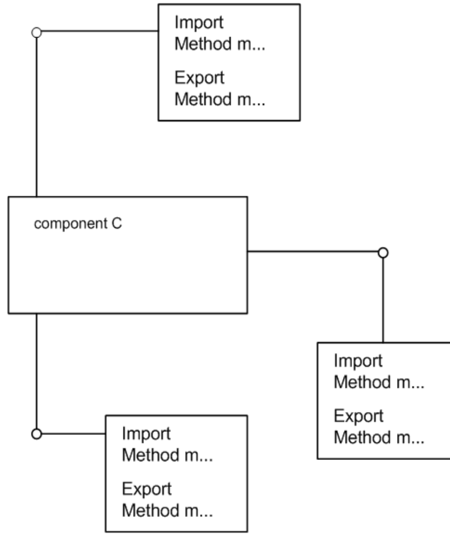



**Figure 6.** Component with three individual export/import interfaces

ber of separate interfaces to components we generalize now export/import interfaces to object-oriented components where a component has a collection of export/import interfaces.

Multiple export/import interfaces have proven to be useful when specifying systems top down, since in this case distinct functionalities may be assigned to a single component without specifying how these tasks are accomplished. Multi-threaded components are helpful when specifying systems with certain response time requirements and multiple users / neighbour systems or when separating complex computation (e.g. search, simulation, remote access) from control interfaces (e.g. GUI). As a consequence, a conventional class as found in object orientation is a special case of a component which is only single-threaded and has only one export/import interface.

**Definition.** Syntactic component interface
A *syntactic component interface* consists of a set syntactic export/import interfaces. □

A component with a set of syntactic export/import interfaces can easily be described graphically as shown in Fig. 6.

We can take an arbitrary export/import interface and turn it into a component by partitioning it into a set of sub-interfaces. On the other hand we always can turn a component into an export/import interface by taking the union over all import interfaces forming one huge import interface and also taking the union over all export interfaces forming one huge export interface. In other words the separation into a number of export/import interfaces provides additional structure but does not lead to more complex models of behaviour.

We now give a first, moderately complex example of a component and its specification.

**Example.** Authorization
We base our specification on a data model for the authorization component. We can do this by an algebraic specification as it was shown for sequences. Since the axioms are

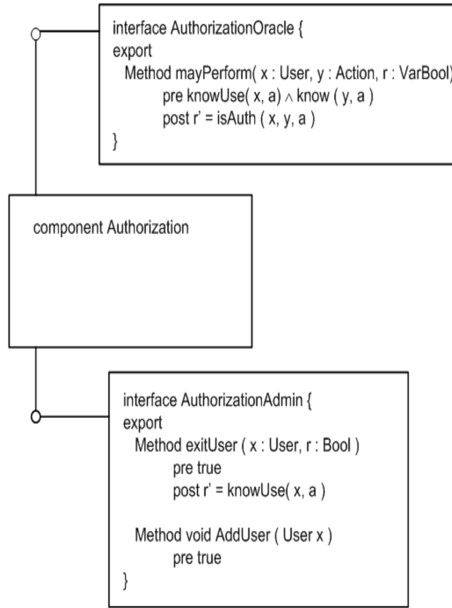



**Figure 7.** Component Authorization

rather straightforward, we skip them.

**SPEC** AUTH =
{ **based_on** BOOL, User, Action

**sort** Auth,

eauth : Auth, "Empty authorization"
addAct : Action, Auth → Auth, *"new action"*
addUse : User, Auth → Auth *"new user"*
addAuth : User, Action, Auth → Auth

knownUse : User, Auth → Bool,
knownAct : Action, Auth → Bool
isAuth : User, Action, Auth → Bool

delAct : Action, Auth → Auth, *"del action"*
delUse : User, Auth → Auth *"del user"*
delAuth : User, Action, Auth → Auth

Auth **generated_by** eauth, addAct, addUse, addAuth

Axioms

...

}

The design by contract for the component based on this specification is directly based on this specification and reads as follows (example due to [19]):

```
Component Authorization {

        a : Var Auth

interface AuthorizationOracle
{
export
method mayPerform (x : User, y : Action, r : Var Bool)
        pre     knownUse(x, a) ∧ knownAct(y, a)
        post    r' = isAuth(x, y, a)
}

interface AuthorizationAdmin
{
export
method existUser(x : User, r : Var Bool)
        pre     true
        post    r' = knownUse(x, a)
method addUser(x : User)
        pre     true
        post    a' = addUse(x, a)
method removeUser(x : User)
        pre     true
        post    a' = delUse(x, a)
method existAction(v : Action, r : Var Bool)
        pre     true
        post    r' = knownAct(x, a)
method addAction(y : Action)
        pre     true
        post    a' = addAct(x, a)
method removeAction(y : Action)
        pre     true
        post    a' = delAct(x, a)
method allow(x : User, y : Action, r : Var Bool)
        pre     knownUse(x, a) ∧ knownAct(y, a)
        post    a' = addAuth(x, y, a) ∧ r' = not isAuth(x, y, a)
method disallow(x : User, y : Action, r : Var Bool)
        pre     true
        post    a' = delAuth(x, y, a) ∧ r' = isAuth(x, y, a)
method getAllowedActions(x : User, actionlist : Var List)
        pre     true
        post    act ∈ actionlist ⇔ knownAct(act, a)
method getAllowedUsers(y : Action, userlist : Var List)
        pre     true
        post    user ∈ userlist ⇔ knownUse(user, a)
}}
```

A graphical representation of the component Authorization is shown in Fig. 7. In this case we work only with an attribute that is global to all interfaces of the component. It is not a problem to introduce attributes that are local to the interfaces. Formally this is just a restricted use of general attributes. □

The specification by contract of a component is like that of an export/import interface as long as we do not consider the structure of forwarded calls and back-calls. The specification expresses that the component works properly and has the indicated effects as long as the environment provides the imported methods as specified. If there is a more complicated situation due to forwarded invocations and back-calls we have to work with the more involved specification techniques such as state machines as discussed above.

In the next section we show how to relate interfaces of components to compose the components.

### 6.3. *Interface Abstraction by Functions on Streams*

So far we have modelled interfaces by state machines. Actually, the state space is thus part of the interface model. At the end there are several state machines modelling interfaces that are observably equivalent, but use different state spaces. One way to relate these state machines are simulations in terms of relations between the state spaces that relate states (or sets of states) with the same traces of pairs of input output in the state transitions. Another possibility is the definition of a mapping that assigns an explicit denotation for a state machine for each state in terms of a function on sequences also called streams. This function is called interface abstraction. It is specified as follows: given a state machine.

$$\Delta : State \times In(c) \rightarrow (State \times Out(c)) \cup \{\bot\}$$

we specify a function

$$\alpha_\Delta : State \rightarrow \left(In(c)^* \rightarrow Out(c)^*\right)$$

by (let $x \in In(c)^*$;by $\langle i \rangle$^x we denote the concatenation of a one element sequence $\langle i \rangle$ with the stream x)

$$(\sigma', O) = \Delta(\sigma, i) \Rightarrow \alpha_\Delta(\sigma)\left(\langle i \rangle^\wedge x\right) = \langle o \rangle^\wedge \alpha_\Delta(\sigma')(x)$$

$$\Delta(\sigma, i) = \bot \Rightarrow \alpha_\Delta(\sigma)\left(\langle i \rangle^\wedge x\right) = \langle\rangle$$

Obviously $\alpha_\Delta(\sigma)$ is prefix monotonic. $\alpha_\Delta(\sigma)$ is the abstract interface for the state machine$(\Delta, i)$, which is the state machine with the initial state $\Delta$. Two classes $c1$ and $c2$ are observably equivalent, if and only if their state machines$(\Delta 1, \sigma 1)$ and $(\Delta 2, \sigma 2)$ fulfil the equation

$$\alpha_{\Delta 1}(\sigma 1) = \alpha_{\Delta 2}(\sigma 2)$$

This definition is rather abstract. However, we get a quite concrete idea of a specification following this idea by specifying equations.

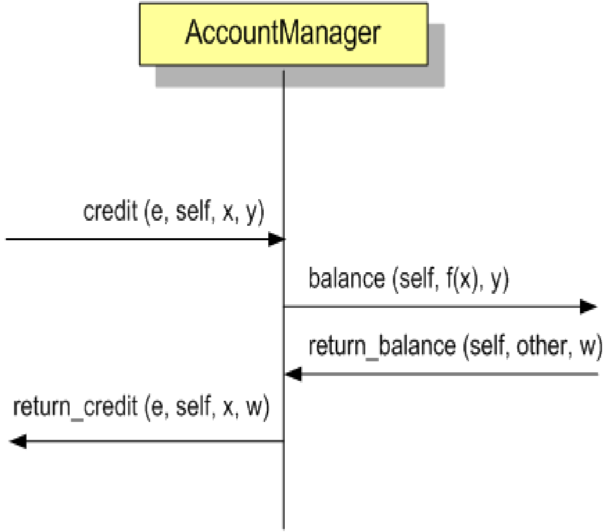



**Figure 8.** Message sequence chart for the Accountmanager

**Example.** Account manager (continued)
We reconsider the simple (class Accountmanager). We define the associated function $\alpha$ by one equation:

$$\alpha\left(\langle credit\,(e, self, x, y, z)\rangle^\wedge \langle return_balance\,(self, other, w)\rangle^\wedge x\right) =$$

$$\langle balance\,(self, f(x), y))\rangle^\wedge \langle return_credit\,(e, self, x, w, f(x))\rangle^\wedge \alpha(x)$$

In this case the specification fairly simple due to the simple structure of the class. In particular, the problem of making the stack explicit disappears.

Note that the sequence chart in Fig. 8 has according to [4] exactly the meaning of the formula above. □

The example shows a line of specifications of export/import interfaces. A better-tuned syntax - for instance special tables - is needed, of course, to make it into a useful technique. Representing object/class behaviours by functions on streams we can use all the specification techniques available for stream process functions (see [3]).

### *6.4. Interface Projection*

We may hide some export and import methods in an interface. This is a way to get simplified versions of a component and simplified views on interfaces. Assume we have a huge component with many interfaces. A large state machine can, in principle, give a precise description of the behaviour of the component in a state based approach. Such a comprehensive behaviour may be rather complex and difficult to understand. Often we are interested in the behaviour of a component with respect to an isolated interface. More

precisely, we assume that all transitions with methods that are not part of that interface are internal and nondeterministic. Given a state machine

$$\Delta: State \times In\,(c) \to (State \times Out\,(c)) \cup \{\perp\}$$

we define the interface projection to the sub-interface $c'$ as follows.We define the nondeterministic state machine (let $c''$ be the export/import interface with $\text{EX}(c'') = \text{IM}(c\backslash c')$ and $\text{IM}(c'') = \text{EX}(c\backslash c')$ ).

$$\Delta'': State \times In\,(c'') \to \wp\,(State \times Out\,(c''))$$

specified by the transition function, that may respond to every call with arbitrary return messages and arbitrary back calls. The machine is highly nondeterministic. We only assume for the machine that it follows the proper scheme of calls and returns (only returns occur for calls that have been issues). Now we define the projection

$$\Delta': State \times In(c') \to \wp((State \times Out(c')) \cup \{ \perp \}$$

by

$$\Delta' = \Delta \parallel \Delta''$$

This definition puts arbitrary input and output on the hidden methods. The idea of hidden state transition proves to be useful also elsewhere. Interface projections are refined by the original interface.

## 7. Composition

In this chapter we finally deal with the composition of components. We are interested in composing two or more components into a composed component. We do that by connecting their interfaces. Due to the typical graphical descriptions of object orientation by diagrams most object oriented methodologies do not consider the composition of classes at all. In this section we show how to compose classes by connecting their interfaces.

One reason for ignoring composition by most of the approaches to object orientation has to do with the fact that imported methods are kept implicit and the behaviour is not described by an open component architecture. Only if we make export/import of methods explicit and use modular specifications, composition is turned into an interesting concept.

### *7.1. Connecting Interfaces*

In this section we study the composition of components by connecting some of their interfaces. This means that we bind imported methods of one component with exported methods of the other one.

In general, we cannot obtain a pure input/output oriented description of a composed system in a straightforward way from the state machines describing the sub-systems that are composed. The reason is that some unbounded chain of method invocations may take

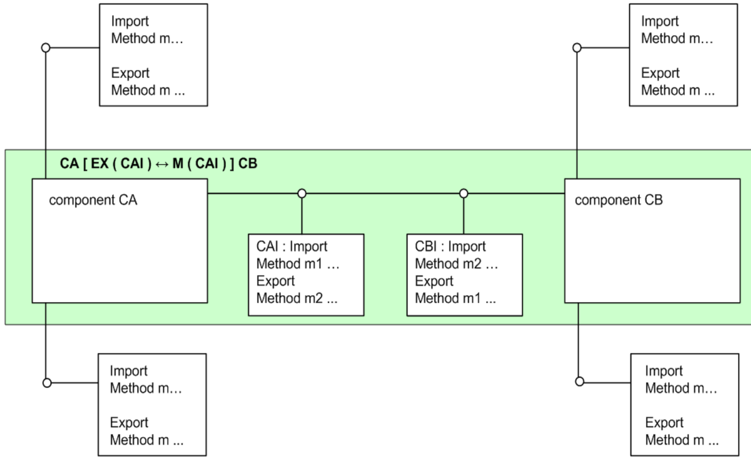



**Figure 9.** Component composition by connection

place between the subsystems within the composed system. This corresponds to state transitions without input or output.

In the following we give a general approach to composing components. In particular, we show how to incorporate all the local state changes.

### 7.1.1. *Syntactic Connection via Interfaces*

To give a syntactic notion of component composition we define the matching of export/import interfaces.

**Definition.** Syntactic Matching export/import interfaces
Given an export/import interface F we call an interface F' matching with F, if IM(F') $\subseteq$ EX(F) and IM(F) $\subseteq$ EX(F'). □

If we have two components CA and CB such that CA has an interface CAI that matches the interface CBI of CB, then we can connect the components via these two interfaces.

**Definition.** Composing two components via matching interfaces
Given two components CA and CB such that CA has an interface CAI that matches the interface CBI, then we can connect these into component denoted by CA[CAI $\leftrightarrow$ CAI]CB. This component has the union of the sets of interfaces of CA and CB as its interface except CAI and CBI. □

Fig. 9 shows the composition of the two components by connecting the matching interfaces. In fact, this definition of composition by connecting interfaces is only syntactic and rather straightforward.

### 7.1.2. *Composing a State Machine from Two by Interface Connection*

If we consider two components CA and CB such that CA has an interface CAI that matches the interface CBI of component CB, then we can connect these. Assume both

components are described by state machines. Let the objects a1 and b2 of CA and CB resp. be given by state machines with state spaces $\Sigma(V1)$ and $\Sigma(V2)$. Let us compose a1 and b2 by connecting their interfaces CAI and CAB. We get a state machine with state space $\Sigma(V1 \cup V2)$.

Fig. 9 shows the composition of components. The construction of the composed state machine is as shown before. More difficult is the derivation of the specification or the design by contract specification for the new state machine out of the given specifications of the original state machines or the original design by contract specification.

The difficulty here is as follows. If we connect the two interfaces there may be a statically unbounded number of method invocations going on between the two components over the connected interface in terms of forwarded method calls and back calls. We speak of "internal chatter". These method invocations correspond to internal state changes after the composition. We need in the general case an invocation stack as part of the state of the new component to be able to determine where to continue after return messages arrive. If we have introduced such a stack as shown in section 5.2, we get internal state transitions that we would like to get rid of. Moreover, we have to characterize the generated states as required in the post-condition while the generated state may be the result of many method invocations going back and forth between the two components.

In technical terms each method invocation at the connected interface results in a state transition in which something is pushed onto the stack and each return results in a state transition where something is popped from the stack.

### *7.2. Other Forms of Composition*

Given a syntactic notion of component in object orientation and a notion of composition the semantically interesting issue is how to model and express composition at the semantic level.

#### *7.2.1. Composition and Semantic Models for Components*

We have considered three concepts for representing the meaning of components independent of code: specification by contract, state machines, and functions of streams of invocation messages.

We have explicitly defined composition for state machines representing behaviours of nondeterministic components. Another idea would be to introduce composition for stream processing functions representing behaviours of nondeterministic components. In fact, this can be defined along the lines of [3].

We do not work out and show this composition explicitly. We only remark that this can be done in a way such that interface abstraction and composition form commuting diagrams - in other words, interface abstraction is a homomorphism for composition on state machines to composition on stream processing functions.

Note that giving a composition in terms of design by contract interface specifications is a more difficult problem. The key problem is to find the pre- and post conditions of the composed component.

#### *7.2.2. Inheritance*

We did not consider, inheritance, at all. There are several reasons for that. First of all, inheritance is mainly interesting for either discussing the type structure of the types that

correspond to classes or, at the code level, for dealing with its effects to inherited code. This allows for methodologically unclean methods such as overwriting of method code in inherited methods.

Another issue of inheritance is the clean description of the type structure (see, for instance, [5]). We are not interested in the type structure in this paper.

The effects of inheritance on the notion of observability, is, however, of some relevance for our theme. An interesting question is, whether inheritance can be seen as a form of composition. In fact, assuming multiple inheritance, where a class can inherit attributes and methods, we may speak of a kind of composition. On one hand, this is semantically not very interesting, as long as this way unions of families of disjoint attribute and method sets are formed. If, however, this way methods are added that allow for new state changes, for instance, then old invariants may become invalid. This poses questions for the specification by contract in the presence of inheritance.

More interesting, however, is the effect of inheritance for observability. By inheritance we get an additional concept of observability that allows us, in principle, to observe all the implementation details of a class - details that should be protected by information hiding.

### 8. Concluding Remarks

Object orientation is a popular programming paradigm that is used quite a lot in practical projects. As we have demonstrated, the methodology for object orientation is still insufficient and incomplete. We demonstrated the limitations and shortcomings of current approaches. Some may be due to the chosen specification concepts. Others are inherent to object orientation and part of this paradigm. We introduced a general notion of component, but still a tractable specification method is not available.

There are a number of directions of research to overcome the described difficulties. One step could be to introduce an explicit notion of component into object oriented languages with a different composition paradigm - such as for instance asynchronous message exchange between components, which encapsulate a family of classes, which cooperate locally by method invocation. Then components form the architecture, which can be hierarchically structured, while the class concepts inside the components represent a detailed design.

Another step would go into a more interaction oriented direction where the cooperation between classes and their objects is specified in terms of method invocation protocols along the lines of assertions on streams as used in FOCUS (see [3]).

## Acknowledgement

It is a pleasure to thank Andreas Rausch, Bernhard Rumpe and Siedersleben for discussions and feedback on topics of this paper. Thanks go also to Andreas Bauer und Tobias Hain for reading drafts.

## References

[1] M. Barnett, R. DeLine, B. Jacobs, M. Fähndrich, K. R. M. Leino, W. Schulte, H. Venter: The Spec# programming system: Challenges and directions. Position paper at VSTTE 2005

[2] M. Broy, C. Hofmann, I. Krüger, M Schmidt: A Graphical Description Technique for Communication in Software Architectures. In: Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97/ICSC'97)

[3] M. Broy, K. Stølen: Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement. Springer 2001

[4] M. Broy: The Semantic and Methodological Essence of Message Sequence Charts. Science of Computer Programming, SCP 54:2-3, 2004, 213-256

[5] M. Broy, M. V. Cengarle, B. Rumpe; Semantics of UML. Towards a System Model for UML. The Structural Data Model, Technische Universität München, Institut für Informatik, Report TUM-IO612, Juni 06

[6] L. de Alfaro, Th. A. Henzinger. Interface-based design. In Engineering Theories of Software-intensive Systems (M. Broy, J. Grünbauer, D. Harel, and C.A.R. Hoare, eds.), NATO Science Series: Mathematics, Physics, and Chemistry, Vol. 195, Springer, 2005, pp. 83-104

[7] D. Herzberg, M. Broy: Modeling layered distributed communication systems. Formal Aspects of Computing 17:1, May 2005, 1-18

[8] C.A.R. Hoare: An Axiomatic Basis for Computer Programming. Comm. ACM 12, 576-583 (1969)

[9] I. Krüger, R. Grosu, P. Scholz, M. Broy: From MSCs to statecharts. In: Proceedings of DIPES'98, Kluwer, 1999

[10] L. Lamport: Specifying Systems: The TLA+ Language and Tools for Harware and Software Engineers. Addison-Wesley Professional 2002

[11] G. T. Leavens, K. R. M. Leino, P. Müller: Specification and verification challenges for sequential object-oriented programs. TR 06-14, Dept. CS, Iowa State University, 2006

[12] Bertrand Meyer: Object-Oriented Software Construction, Prentice Hall, 1988

[13] Bertrand Meyer: Applying "Design by Contract ", in Computer (IEEE), 25, 10, October 1992, pages 40-51

[14] P. Müller, A. Poetzsch-Heffter: Modular Specification and Verification Techniques for Object-Oriented Software Components. In: Leavens, G. T. and Sitaraman, M. ed., Foundations of Component-Based Systems. Cambridge University Press 2000

[15] Oscar Nierstrasz: ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, Jul 26-30, 1993, Proceedings Springer 1993

[16] D. Parnas: On the criteria to be used to decompose systems into modules. Comm. ACM 15, 1972, 1053-1058

[17] A. Poetzsch-Heffter: Specification and Verification of Object-Oriented ProgramS. Habilitation thesis, Technical University of Munich, Available, January, 1997

[18] B. Selic, G. Gullekson. P.T. Ward: Real-time Objectoriented Modeling. Wiley, New York 1994

[19] J. Siedersleben: Moderne Software-Architektur. Umsichtig planen, robust bauen mit Quasar. Dpunkt Verlag, August 2004

[20] M. Spivey: Understanding Z - A Specification Language and Its Formal Semantics. Cambridge Tracts in Theoretical Computer Science 3, Cambridge University Press 1988

[21] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language for Object-Oriented Development, Version 1.0, RATIONAL Software Cooperation

[22] P. Wegner, S.B. Zdonik: Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In Proceedings ECOOP'88, ed. S. Gjessing and K. Nygaard, Lecture Notes in Computer Science 322, Springer-Verlag, Oslo, Aug. 15-17, 1988, 55-77

[23] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy: On hierarchies of abstract data types. Technische Universität München, Institut für Informatik, TUM-18007, May 1980. Revidierte Fassung: Acta Informatica 20, 1983, 1-33

[24] P. Zave, M. Jackson: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, January 1997

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Using Invariants to Reason About Cryptographic Protocols

Ernie COHEN
*Microsoft Corporation, Redmond, USA*

**Abstract.** This tutorial describes how to reason about cryptographic protocols in perfect cryptography models using ordinary program invariants.

## 1. Introduction

If there is a lesson to be learned from the last 30 years of concurrent programming research, it is that the most effective way to prove safety properties of most concurrent programs is to use global invariants, and that improvements to concurrent programming reasoning methodology come from finding better ways to structure these invariants.

A good example is the analysis of cryptographic protocols in perfect cryptography models. For many years, the research community virtually ignored invariance reasoning, wandering through dead-ends such as special-purpose epistemic logics. When hitherto undiscovered bugs finally drove the community back to operational models, the first invariance proofs were initially difficult, requiring complex, recursive invariants that were specific to the particular property being proved[6,5]. A major simplification came with the discovery of a new invariant structure that simultaneously took into account the whole system state[2].

In this tutorial, we present a simple way to use ordinary program invariants to reason about cryptographic protocols. Of course, few readers need to do such reasoning. We present the tutorial because there is a tendency nowadays to thing of invariants as big, unstructured set of states that should be generated by finite-state exploration. In fact, a suitable invariant structure is the key to effective reasoning in any new program domain.

## 2. Cryptographic Protocols

Cryptographic protocols can be viewed as distributed programs with three distinguishing features:

- they are designed to operate in a hostile communication environment, where an adversary with certain computational abilities controls the communication medium;
- they make essential use of secrets, typically created through random number generation;

- they use encryption to hide these secrets from the adversary, while revealing them to appropriate participants.

As an example, we'll take what has become a standard benchmark in the field, the Needham-Schroeder-Lowe public key[1] authentication protocol [4]:

$A \rightarrow B$: $\{A, Na\}_{k(B)}$
$B \rightarrow A$: $\{B, Na, Nb\}_{k(A)}$
$A \rightarrow B$: $\{Nb\}_{k(B)}$

Following traditional protocol notation, tuples are enclosed in curly braces and subscripting denotes encryption. Here, $A$ and $B$ represent names of *principals* (protocol participants), and $Na$ and $Nb$ represent *nonces* (freshly generated random numbers), generated by $A$ and $B$, respectively.

The steps above describe a typical sequence of messages sent in a single protocol session. There can be any number of such sessions going on at the same time. The purpose of this protocol is to allow $A$ and $B$ to securely exchange nonces (typically to be used to generate a shared secret key for subsequent communication). In the first step, $A$ generates a nonce $Na$, tuples it together with his name, encrypts the tuple under $B$'s public key, and sends the message to $B$. When $B$ receives the message, he decrypts it, generates his own nonce $Nb$, tuples it together with his name and $Na$, encrypts the tuple under $A$'s public key, and sends the message back to $A$. When $A$ receives the reply, he decrypts the message, checks that the $Na$ value agrees with the value he sent in his first message, and sends the $Nb$ value from the message back to $B$, encrypted under $B$'s public key. When $B$ receives this message, he checks that the $Nb$ value agrees with the value he sent in the second step, and accepts the protocol as finished.

This protocol is designed to operate in parallel with an adversary that can copy, delete, reorder, or redirect any messages. In addition, he can form new messages by encrypting messages that he has seen under keys that he has seen, and can decrypt a message he has seen if he has also seen a suitable decryption key. The properties we would like to prove for the protocol are

- if $A$ completes the third protocol step successfully, then either $B$ has completed the second protocol step with corresponding values for $A$, $Na$ and $Nb$, or one of $A$ or $B$ is compromised (i.e., his private key is available to the adversary);
- if $B$ accepts the protocol as completed, then either $A$ has completed the third protocol step with corresponding values for $B$, $Na$, and $Nb$, or one of $A$ or $B$ is compromised.

Other properties, such as the conditions under which $Na$ and $Nb$ remain secret, are proved in the course of establishing suitable invariants for the NSL protocol.

## 3. Modeling the Protocol

In order to reason about the protocol above, we must first translate it into a transition system. This means we have to model both the state space and the transition relation. Mod-

[1] In fact, we formulate the protocol so that the type of key used is irrelevant to the correctness proof.

eling the transition relation means modeling both the actions of the protocol participants and the actions of the adversary.

### 3.1. Encryption

Consider first the situation where $B$ has received the first message sent by $A$. We want to make sure that the values for $A$ and $Na$ he obtains by decrypting the message agree with the values used to generate the message. The usual way to allow such reasoning is to assume that encryption is injective in all of its arguments. Similarly, we would expect normally that principals do not share public keys. We can formalize these as axioms on the message space; we refer to these axioms jointly as "injectivity":

$$\{\vec{X}\}_Y = \{\vec{U}\}_V \Rightarrow \vec{X} = \vec{U} \wedge Y = V$$
$$k(X) = k(Y) \Rightarrow X = Y$$

When writing formulas, ordinary identifiers starting with uppercase letters (such as $Y$ and $V$ above) are variables ranging over messages (the data values manipulated by principals and sent between principals); variables written with vector notation (such as $\vec{X}$ and $\vec{U}$ above) range over tuples of message values. Both kinds of variables are implicitly universally quantified when they appear free in a formula. We consider two tuples to be equal if they have equal length and corresponding components are equal.

We need to model when the adversary can decrypt an encrypted message. This depends on the kind of cryptography used for the encryption. To handle many flavors of cryptography at once, we represent the relation between encryption and decryption keys with a binary relation $d(X, Y)$; intuitively, $d(X, Y)$ ("$X$ decrypts for $Y$") means that messages encrypted under key $Y$ can be decrypted with key $X$. For example, we could define symmetric keys, key pairs, and cryptographic hash functions with the axioms

$$sk(X) \Leftrightarrow (\forall Y : d(Y, X) \Leftrightarrow X = Y)$$
$$kp(X, Y) \Leftrightarrow X \neq Y \wedge (\forall Z : (d(Z, X) \Leftrightarrow Z = Y) \wedge (d(Z, Y) \Leftrightarrow Z = X))$$
$$hash(X) \Leftrightarrow (\forall Y : \neg d(Y, X))$$

These say that (1) a key is symmetric if it is the only key that decrypts for itself, (2) a pair of keys is a key pair if the two keys are the only keys that decrypt for each other, and (3) encryptions under a cryptographic hash function cannot be decrypted.

We say that a message is *atomic* if it is not in the range of encryption:

$$atom(X) \Leftrightarrow (\forall \vec{Y}, Z : X \neq \{\vec{Y}\}_Z)$$

We distinguish atoms because some operations (like random nonce generation) should not produce encryptions.

### 3.2. The State Space

As usual, the state of a distributed system can be obtained by composing the state of the principals along with the state of the communication medium.

The usual way to model the communication state of a distributed program is with a queue of messages between each pair of communicating principals. However, because

the adversary controls all message delivery, there is little point in keeping track of what order messages were sent (since the adversary can reorder messages), how many times a sent message was sent (since the adversary can duplicate or delete messages), or even where a message was sent from/to (since the adversary can move messages between channels). This means that the relevant communication state is given by the set of all messages that have ever been sent (by anybody). We can represent this state by a single state predicate $pub(X)$ ("$X$ is published"). Thus, sending a message is just publishing it, and receiving a message is just checking that it has been published.

Next, consider the state of the adversary. The adversary can construct new messages from the messages that he has seen or constructed. Rather than trying to track these separately, we might as well assume that the adversary has seen every published message, and that he publishes every message that he constructs. Thus, the adversary simply constructs new messages from previously published messages, and publishes the results.

Finally, consider the state of a principal. The obvious model is a set of protocol sessions, each recording the history of the session. However, this approach has some drawbacks; many protocols include actions that are not conveniently associated with a single protocol session (for example, generating a long-term key to be shared between two principals), or have a structure that is not conveniently broken up into sessions. So instead, we just record what protocol actions have been executed, without trying to group the actions into sessions. (We are on safe ground in doing this, because we could always introduce additional session identifiers into the individual protocol actions.) Formally speaking, we introduce a *history predicate* for each protocol action; the arguments to each predicate capture the relevant state of the principal performing the action.

For example, in the NSL protocol, we can record instances of the first protocol action with a predicate $p0(A, Na, B)$; for any messages $A$,$Na$, and $B$, this predicate is true iff principal $A$ has executed the first protocol step, intending to communicate with $B$, and generating nonce value $Na$. Similarly, we can record instances of the second and third steps with state predicates $p1(A, B, Na, Nb)$ and $p2(A, B, Na, Nb)$. Finally, we can record $B$ accepting the final protocol message with a state predicate $p3(A, B, Na, Nb)$.

The state of the whole system is given by the values assigned to the state predicates $pub$,$p0$,$p1$,$p2$, and $p3$. By the "value" of a state predicate, we mean an interpretation in the usual sense of logic, i.e. a state assigns to each state predicate a set of tuples for which the predicate is true.

Since state predicates record history, they are monotonically weakening during protocol execution. (Published messages are never unpublished, executed protocol steps are never unexecuted.) We say a formula is *positive* iff state predicates appear in the formula only with positive polarity (i.e., governed by an even number of negations). It follows that every positive formula is *stable* - if it holds in a state, it also holds in all subsequent states. This property simplifies program reasoning considerably.

### 3.3. Actions

We specify the transitions of the model with actions of the form

$$guard \longrightarrow terms$$

where $guard$ is a formula and $terms$ is a list of state predicates applied to terms. This action is executed by nondeterministically choosing an arbitrary message value for each

free variable in the action, such that $guard$ is true in the current state, and adding a minimal set of tuples to the interpretations of the state predicates to make each of the formulas of $terms$ true. For example, the action

$$pub(A) \wedge pub(B) \wedge atom(Na) \longrightarrow p0(A, B, Na)$$

is executed by choosing arbitrary values for $A$, $B$ and $Na$ such that $A$ and $B$ are published and $Na$ is an atom, and adding the corresponding $(A, B, Na)$ tuple to $p0$.

### *3.4. Adversary Actions*

The adversary can

- publish a new random nonce (adversary nonce generation);
- tuple together a number of published messages, encrypt them under a published key, and publish the result (adversary encryption);
- publish $\vec{X}$, if $\{\vec{X}\}_Y$ is published and there is a published key that decrypts for $Y$ (adversary decryption).

Consider first random nonce generation. The NSL protocol suggests two requirements:

- No random nonce value should be generated more than once. Otherwise, the adversary might luckily generate some nonce that was being kept as a secret, and use it to compromise the protocol just as if he knew the secret. In other words, the newly generated value must be *fresh*.
- Randomly generated nonces should not collide with encryptions. Otherwise, the adversary could break the protocol by just randomly generating the second protocol message, fooling $A$ into thinking that $B$ had executed his first step. In other words, the newly generated value must be *atomic*.

We can define freshness as follows. A fresh value should not coincide with a nonce value that has been generated either for $Na$ in the first protocol step or for $Nb$ in the second step, nor should a fresh value be published. (This latter requirement guarantees that it doesn't collide with nonces previously generated by the adversary.) In other words, we should have the properties

(1) $fresh(X) \Rightarrow \neg p0(A, B, X)$

(2) $fresh(X) \Rightarrow \neg p1(A, B, Na, X)$

(3) $fresh(X) \Rightarrow \neg pub(X)$

(Note that freshness will normally be state dependent, but is not stable.) We can then specify adversary nonce generation with the action

$$fresh(X) \wedge atom(X) \longrightarrow pub(X)$$

Adversary encryption translates directly to

$$pub(\vec{X}) \wedge pub(Y) \longrightarrow pub(\{\vec{X}\}_Y)$$

Here, as below, $pub(\vec{X})$ means that every component of the tuple $X$ is published.

Define $dk(X)$ ("$X$ is a decryptable key") iff some published key decrypts for $X$:

$$dk(X) \Leftrightarrow (\exists\, Y : d(Y, X) \wedge pub(X))$$

We can then model adversary decryption with the action

$$pub(\{\vec{X}\}_Y) \wedge dk(Y) \longrightarrow pub(\vec{X})$$

### *3.5. Protocol Actions*

Finally, we turn to formalizing the actions of the protocol itself. To save writing the same message terms over and over, let's define some macros:

$$m0 = \{A, Na\}_{k(B)}$$
$$m1 = \{B, Na, Nb\}_{k(A)}$$
$$m2 = \{Nb\}_{k(B)}$$

The obvious way to formalize the first step is with the action

$$fresh(Na) \wedge atom(Na) \longrightarrow p0(A, B, Na), pub(m0)$$

However, this does not constrain $A$ in any way. For example, we could choose for $A$ a value that contains some arbitrary secret, which would obviously break the protocol. So we must constrain $A$ (at least) to not leak any new information. The simplest way to make sure that a piece of data cannot leak information is to make sure that it is already published:

$$fresh(Na) \wedge atom(Na) \wedge pub(A) \longrightarrow p0(A, B, Na), pub(m0)$$

For the second step, we have to constrain $B$ (since it was not constrained in the first step). As expected, we model the receipt of $m0$ by checking that it is published:

$$pub(m0) \wedge fresh(Nb) \wedge atom(Nb) \wedge pub(B) \longrightarrow p1(A, B, Na, Nb), pub(m1)$$

For the third step, $A$ has to check that we've already started a "session" with appropriate values for $B$, and $Na$; similarly, before finishing the protocol, $B$ must check that the $Nb$ value he receives corresponds to the value he sent in the second step:

$$\begin{aligned} p0(A, B, Na) \wedge pub(m1) &\longrightarrow p2(A, B, Na, Nb), pub(m2) \\ p1(A, B, Na, Nb) \wedge pub(m2) &\longrightarrow p3(A, B, Na, Nb) \end{aligned}$$

As it turns out, we almost always use state predicates with the exact arguments given above. To reduce unnecessary writing, we use the convention that whenever one of the state predicates $p0 \ldots p3$ appears with no arguments, it implicitly applies to the default sequence of arguments given above. (To take care of cases where we want a history predicate applied to a second set of arguments, $p0'$ abbreviates $p0(A', B', Na')$, and similarly for the other history predicates.) Thus, we can rewrite the whole model more succinctly as follows:

$$\begin{array}{ll}
\mathit{fresh}(X) \wedge \mathit{atom}(X) & \longrightarrow \mathit{pub}(X) \\
\mathit{pub}(\vec{X}) \wedge \mathit{pub}(Y) & \longrightarrow \mathit{pub}(\{\vec{X}\}_Y) \\
\mathit{pub}(\{\vec{X}\}_Y) \wedge \mathit{dk}(Y) & \longrightarrow \mathit{pub}(\vec{X}) \\
\mathit{fresh}(Na) \wedge \mathit{atom}(Na) \wedge \mathit{pub}(A) & \longrightarrow p0, \mathit{pub}(m0) \\
\mathit{pub}(m0) \wedge \mathit{fresh}(Nb) \wedge \mathit{atom}(Nb) \wedge \mathit{pub}(B) & \longrightarrow p1, \mathit{pub}(m1) \\
p0 \wedge \mathit{pub}(m1) & \longrightarrow p2, \mathit{pub}(m2) \\
p1 \wedge \mathit{pub}(m2) & \longrightarrow p3
\end{array}$$

While this kind of abbreviation is not terribly important for a toy protocol like this one, real protocols often have state predicates with many arguments, making explicit parameter lists painful and error prone. (A more sophisticated renaming scheme can be found in [2].)

Note that there are no explicit actions for publishing the principal names. We can think of principal names (as well as other public data values) as simply being generated by adversary nonce generation.

## 4. Structural Invariants

Before attacking the protocol proper, we consider two kinds of invariants common to all such protocols.

### *4.1. Unicity Invariants*

Recall that we require that nonces are fresh when they are generated, but that freshness is not stable. Therefore, we need to capture the essence of fresh nonce generation with invariants. Intuitively, the properties we want are that

- Two different $p0$ steps cannot generate the same $Na$ nonce.
- Two different $p1$ steps cannot generate the same $Nb$ nonce.
- The same nonce cannot be generated as both an $Na$ nonce and an $Nb$ nonce.

Note that these depend only on which protocol steps generate fresh nonces, not on any particular properties of the protocol itself.[2] We can formalize these properties as the following invariants:

[2] In a tool, instead of writing the definition of fresh atoms in the guard of an action, we would build fresh nonce generation into the syntax, so that these lemmas could be generated automatically.

$$p0 \wedge p0' \wedge Na = Na' \Rightarrow A = A' \wedge B = B'$$
$$p1 \wedge p1' \wedge Nb = Nb' \Rightarrow A = A' \wedge B = B' \wedge Na = Na'$$
$$p0 \wedge p1' \Rightarrow Na \neq Nb'$$

We refer to these invariants generically as "unicity". To see that these are invariants, notice that they hold in the initial state, since the left-hand sides are false (since all history predicates are initially false). The first two protocol steps preserve these properties because of the freshness conjuncts in the guards. These last two steps and the adversary steps preserve these because they do not modify $p0$ or $p1$.

In practice, we use the following (equivalent) formulation of unicity. Suppose $\mathsf{f}$ and $\mathsf{g}$ are formulas, and $\mathsf{v}$ is a list of variables that does not include $Na$. Then

$$\mathsf{f} \wedge p0 \wedge (\exists\, \mathsf{v}, A, B : p0 \wedge \mathsf{g}) \Rightarrow \mathsf{f} \wedge p0 \wedge (\exists\, \mathsf{v} : \mathsf{g})$$

is an invariant of the system. In other words, if we have $p0$ as a conjunct both inside and outside of an existential quantification, and the quantified variables do not include $Na$, we can remove $A$ and $B$ from the list of existentially quantified variables. Similarly, if $\mathsf{v}$ does not include $Nb$,

$$\mathsf{f} \wedge p1 \wedge (\exists\, \mathsf{v}, A, B, Na : p1 \wedge \mathsf{g}) \Rightarrow \mathsf{f} \wedge p1 \wedge (\exists\, \mathsf{v} : \mathsf{g})$$

is also an invariant of the system.

### *4.2. Guard Invariants*

If we remove the freshness conjuncts from the protocol transitions, the history predicate updated by each transition implies its corresponding guard. That is, we have the invariants

$$p0 \Rightarrow atom(Na) \wedge pub(A)$$
$$p1 \Rightarrow atom(Nb) \wedge pub(B) \wedge pub(m0)$$
$$p2 \Rightarrow p0 \wedge pub(m1)$$
$$p3 \Rightarrow p1 \wedge pub(m2)$$

For every binding of the free variables, of these formulas is an invariant. To see why, note that each formula holds initially, because all history predicates are initially $false$. An implication can be falsified only by a step that truthifies the hypothesis or falsifies the consequent. The only transitions that truthify the hypotheses simultaneously check that the consequent holds, and no step falsifies the consequent (because the consequent is positive, hence stable). Note that we have to remove the freshness conjuncts because freshness is not stable.

We reference these invariants with the hints $p0$, $p1$, etc.

## 5. The secrecy invariant

The invariants so far do not constrain the messages that can be published; they could be satisfied even if all messages were published. Since messages are the only way a principal can learn about the states of other principals, we need an invariant that constrains the conditions under which messages can be published.

The obvious way to write such an invariant is in the form

$$pub(X) \Rightarrow ok(X)$$

where $ok(X)$ is a disjunction of cases, one for each way that a message can be published. However, there is a problem with this approach, because of the adversary decryption case; this case would produce a disjunct of the form

$$\vee\ (\exists\, Y : pub(\{X\}_Y) \wedge dk(Y))$$

Regardless of any other disjunctions, this would allow the possibility that *all* messages are published (as long as some key is decryptable). This would make the secrecy invariant practically useless.

Thus, we must find an indirect way to eliminate the adversary decryption case from the secrecy invariant. Of course, we can't simply ignore the adversary decryption action, because the set of cases must be closed under all actions. Instead, we add additional disjuncts to the definition of $ok$ to make $ok$ closed under adversary decryption.

So let's construct the disjuncts of $ok(X)$ for NSL:

- For each protocol step that publishes a message, there is a case corresponding to the published message; in each case, we know that the corresponding history predicate also holds (since it is truthified when the message is published):

$$\vee\ (\exists\, A, Na, Nb : X = m0 \wedge p0)$$
$$\vee\ (\exists\, A, B, Na, Nb : X = m1 \wedge p1)$$
$$\vee\ (\exists\, A, Na, Nb : X = m2 \wedge p2)$$

- There is a case for a message that arose through adversary encryption (of any arity); in this case, we know that the components of the message and the encryption key are already published:

$$\vee\ (\exists\, \vec{Y}, Z : X = \{\vec{Y}\}_Z \wedge pub(\vec{Y}) \wedge pub(Z))$$

- There is a case for adversary nonce generation; in this case, we know that the generated nonce is not generated by one of the ordinary protocol steps. Defining

$$junk(X) \Leftrightarrow atom(X) \wedge (\forall A, B, Na, Nb : (p0 \Rightarrow X \neq Na) \wedge (p1 \Rightarrow X \neq Nb))$$

  we have a case

$$\vee\ junk(X)$$

These cases cover all of the actions except for adversary decryption. To close the set of cases under adversary decryption, we need to consider what new cases can be generated from each of the message cases we have already.

Obviously, decryption cannot produce a new case from those cases that produce atoms (by the definition of atonicity). Moreover, decryption of a message produced under the adversary encryption case doesn't produce anything new, because all of the tupled messages were already published. Thus, we need only consider adversary decryption of messages published from honest protocol actions. These actions may result in the publication of previously unpublished nonce values.

Looking at the protocol, it seems that $Na$ or $Nb$ nonces should be revealed only if $dk(k(A)) \vee dk(k(B))$. (Heuristically, this is because $k(A)$ and $k(B)$ are the only keys that guard the exposure of messages that mention $A$ and $B$.) Thus, our last two cases are

$$\vee\ X = Na \wedge p0 \wedge (dk(k(A)) \vee dk(k(B)))$$
$$\vee\ X = Nb \wedge p1 \wedge (dk(k(A)) \vee dk(k(B)))$$

Putting all of this together, we define $ok(X)$ by

$$\begin{aligned} ok(X) \Leftrightarrow (\exists \quad & A, B, Na, Nb, \vec{Y}, Z : \\ & \vee (X = m0 \wedge p0) \\ & \vee (X = m1 \wedge p1) \\ & \vee (X = m2 \wedge p2) \\ & \vee (X = \{\vec{Y}\}_Z \wedge pub(\vec{Y}) \wedge pub(Z)) \\ & \vee\ junk(X) \\ & \vee (X = Na \wedge p0 \wedge (dk(k(A)) \vee dk(k(B)))) \\ & \vee (X = Nb \wedge p1 \wedge (dk(k(A)) \vee dk(k(B))))) \end{aligned}$$

Note that we haven't yet proved that the secrecy invariant is actually an invariant.

### 5.1. Consequences of the Secrecy Invariant

Using the secrecy invariant, we can conclude information about the state from publication of a message. For example, if $pub(m0)$, then $m0$ must match one of the cases in the secrecy invariant. By atomicity and injectivity, the only cases that can be matched are the $p0$ case and the adversary encryption case. (Similarly for the other message forms.) Hence, by injectivity, we have the consequences

$$pub(m0) \Rightarrow p0 \vee (pub(k(B)) \wedge pub(A) \wedge pub(B))$$
$$pub(m1) \Rightarrow p1 \vee (pub(k(A)) \wedge pub(B) \wedge pub(Na) \wedge pub(Nb))$$
$$pub(m2) \Rightarrow (\exists\ A, Na : p2) \vee (pub(k(B)) \wedge pub(Nb))$$

We reference these consequences by the hints "$m0$", "$m1$", and "$m2$", respectively.

Similarly, thanks to the unicity invariants and the secrecy invariant, we have the consequences

$$p0 \wedge pub(Na) \Rightarrow dk(k(A)) \vee dk(k(B))$$

$$p1 \wedge pub(Nb) \Rightarrow dk(k(A)) \vee dk(k(B))$$

We reference these consequences with the hints "$Na$" and "$Nb$", respectively.

We repeat that these are mere consequences of the secrecy invariant; they are guaranteed to hold only in states where we know the secrecy invariant holds.

### 5.2. Checking the Secrecy Invariant

The secrecy invariant trivially holds in the initial state, since initially no messages are published. Because the definition of $ok$ is positive, $ok$ messages remain $ok$. Thus, to show that the secrecy invariant is an invariant, it suffices to show that messages are $ok$ on the step before they are published. The only transition for which this is nontrivial is adversary decryption. Because atoms cannot be decrypted, the proof obligations are as follows (all assuming the secrecy invariant):

$$\begin{array}{l}
p0 \wedge dk(k(B)) \Rightarrow ok(A) \\
p0 \wedge dk(k(B)) \Rightarrow ok(Na) \\
p1 \wedge dk(k(A)) \Rightarrow ok(B) \\
p1 \wedge dk(k(A)) \Rightarrow ok(Na) \\
p1 \wedge dk(k(A)) \Rightarrow ok(Nb) \\
p2 \wedge dk(k(B)) \Rightarrow ok(Nb)
\end{array}$$

We can discharge these obligations as follows (the hint $ok$ references the definition of $ok$):

$$\begin{array}{ll}
p0 \wedge dk(k(B)) & \Rightarrow \{p0\} \\
pub(A) & \Rightarrow \{inv\} \\
ok(A) &
\end{array}$$

$$\begin{array}{ll}
p0 \wedge dk(k(B)) & \Rightarrow \{ok\} \\
ok(Na) &
\end{array}$$

$$\begin{array}{ll}
p1 \wedge dk(k(A)) & \Rightarrow \{p1\} \\
pub(B) & \Rightarrow \{inv\} \\
ok(B) &
\end{array}$$

$$\begin{array}{ll}
p1 \wedge dk(k(A)) & \Rightarrow \{p1\} \\
pub(m0) \wedge dk(k(A)) & \Rightarrow \{m0\} \\
(p0 \vee pub(Na)) \wedge dk(k(A)) & \Rightarrow \{\text{logic}\} \\
(p0 \wedge dk(k(A))) \vee pub(Na) & \Rightarrow \{ok\} \\
ok(Na) \vee pub(Na) & \Rightarrow \{inv\} \\
ok(Na) &
\end{array}$$

$$
\begin{array}{l}
p1 \wedge dk(k(A)) \Rightarrow \{ok\} \\
ok(Nb)
\end{array}
$$

$$
\begin{array}{ll}
p2 \wedge dk(k(B)) & \Rightarrow \{p2\} \\
pub(m1) \wedge dk(k(B)) & \Rightarrow \{m1\} \\
(p1 \vee pub(Nb)) \wedge dk(k(B)) & \Rightarrow \{\text{logic}\} \\
(p1 \wedge dk(k(B))) \vee pub(Nb) & \Rightarrow \{ok\} \\
ok(Nb) \vee pub(Nb) & \Rightarrow \{inv\} \\
ok(Nb) &
\end{array}
$$

This concludes the proof that the secrecy invariant is, in fact, an invariant of the transition system.

## 6. Proving Authentication Properties

Now that we have established the secrecy invariant is an invariant, we can prove other invariant properties of the protocol by ordinary logical reasoning from the secrecy, unicity, and guard invariants. This is because we have intentionally chosen to make the secrecy invariant as strong as possible.

Here are proofs of the authentication properties for the NSL protocol. The first theorem says that, if $A$ completes his final step, then either $A$ or $B$ is compromised, or $B$ has completed his first step, with the same values for $A$,$B$,$Na$ and $Nb$:

$$
\begin{array}{ll}
p2 & \Rightarrow \{p2\} \\
p0 \wedge pub(m1) & \Rightarrow \{m1\} \\
p0 \wedge (p1 \vee pub(Na)) & \Rightarrow \{\text{logic}\} \\
(p0 \wedge pub(Na)) \vee p1 & \Rightarrow \{Na\} \\
dk(k(A)) \vee dk(k(B)) \vee p1 &
\end{array}
$$

The second theorem says that, if $B$ completes his final step, then either $A$ or $B$ is compromised, or $A$ has completed his final step, with the same values for $A$,$B$,$Na$ and $Nb$:

$$
\begin{array}{ll}
p3 & \Rightarrow \{p3\} \\
p1 \wedge pub(m2) & \Rightarrow \{m2\} \\
p1 \wedge ((\exists\, A, Na : p2) \vee pub(Nb)) &
\end{array}
$$

Notice that the term we want in the conclusion (namely, $p2$) appears, but enclosed in an existential quantification. Note, however, that we have $p1$ outside of the quantification, and the quantification does not quantify over $Nb$. Therefore, if we can get $p1$ as a conjunct inside the quantification, we can strip away the quantification of $A$ and $Na$, using the unicity lemma for $Nb$. This motivates the following continuation of the proof, where we expand out the cases under which $p2$ can arise:

$$
\begin{array}{ll}
p1 \wedge ((\exists\, A, Na : p2) \vee pub(Nb)) & \Rightarrow \{p2 \qquad\} \\
p1 \wedge ((\exists\, A, Na : p2 \wedge pub(m1)) \vee pub(Nb)) & \Rightarrow \{m1 \qquad\} \\
p1 \wedge ((\exists\, A, Na : p2 \wedge (p1 \vee pub(Nb))) \vee pub(Nb)) & \Rightarrow \{\text{logic} \qquad\} \\
p1 \wedge ((\exists\, A, Na : p2 \wedge p1) \vee pub(Nb)) & \Rightarrow \{\text{logic} \qquad\} \\
(p1 \wedge pub(Nb)) \vee (p1 \wedge (\exists\, A, Na : p2 \wedge p1)) & \Rightarrow \{Nb \text{ unicity}\} \\
(p1 \wedge pub(Nb)) \vee (p1 \wedge p2) & \Rightarrow \{Nb \qquad\} \\
dk(k(A)) \vee dk(k(B)) \vee p2 &
\end{array}
$$

## 7. Exercises

1. Show that the NSL protocol still works without the assumption that $k$ is injective.
2. Remove $B$ from $m1$ and try to prove the resulting protocol correct. Where does the proof get stuck? Can you find a counterexample that shows the new protocol is broken?
3. Replace the keying function with new protocol actions that generate public/private key pairs for principals, publishing the public key. Show that the resulting protocol is still correct.
4. Model the compromise of principals explicitly, by introducing a new state predicate that tracks which principals have been compromised. Add suitable actions to model the compromise of a principal and to release his information to the adversary. If necessary, reformulate and prove the safety properties of the new protocol.
5. Add to NSL an action that receives an arbitrary message and simply republishes it. Try to treat it like the other protocol actions (i.e., generate corresponding guard and message lemmas, and add a corresponding case to the secrecy invariant). Does this action affect your ability to reason about the protocol? Can you modify your method to prove the protocol correct?
6. Add to NSL an adversary action that permutes the component order of encrypted 2-tuples. Prove the resulting protocol correct.
7. Model and verify a protocol with two levels of encryption. How does the proof structure change?
8. Model and verify a "repeated authentication" protocol, such as Kao-Chow.
9. How would you model a key $K$ with a pair of decryption keys both decryption keys are required to decrypt messages encrypted under $K$?
10. Try to design your own crypto protocol, specify it, and prove it correct.
11. [Research] Modify the methodology here to work with more realistic protocol models. For example, model and reason about any of the following:
    - Vernam encryption (bitwise exclusive-or)
    - bitwise concatenation and projection
    - encryption that is injective with respect to the key and with respect to the encrypted message, but not necessarily both together.

    Some guidance for these can be found in [3].
12. [Research] Construct a simple protocol for unidirectional secret communication, and use simulation relations to prove that it simulates a protocol where the adversary sees nothing but noise. (See [1] for a process-algebraic solution to this challenge.)

## References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[2] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, May 2003.

[3] E. Cohen. Taps: The last few slides. In *FASec 2002: Formal Aspects of Security*, volume 2629 of *LNCS*, pages 183–190. Springer, 2003.

[4] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, 1996.

[5] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[6] L. Paulson. The inductive approach to verifying cryptographic protocols. *JCS*, 6:85–128, 1998.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Verified Interoperable Implementations of Security Protocols

Karthikeyan BHARGAVAN [a], Cédric FOURNET [a], Andrew D. GORDON [a], and Stephen TSE [b]

[a] *Microsoft Research*
[b] *University of Pennsylvania*

**Abstract.** We present an architecture and tools for verifying implementations of security protocols. Our implementations can run with both concrete and symbolic implementations of cryptographic algorithms. The concrete implementation is for production and interoperability testing. The symbolic implementation is for debugging and formal verification. We develop our approach for protocols written in F#, a dialect of ML, and verify them by compilation to ProVerif, a resolution-based theorem prover for cryptographic protocols. We establish the correctness of this compilation scheme, and we illustrate our approach with protocols for Web Services security.

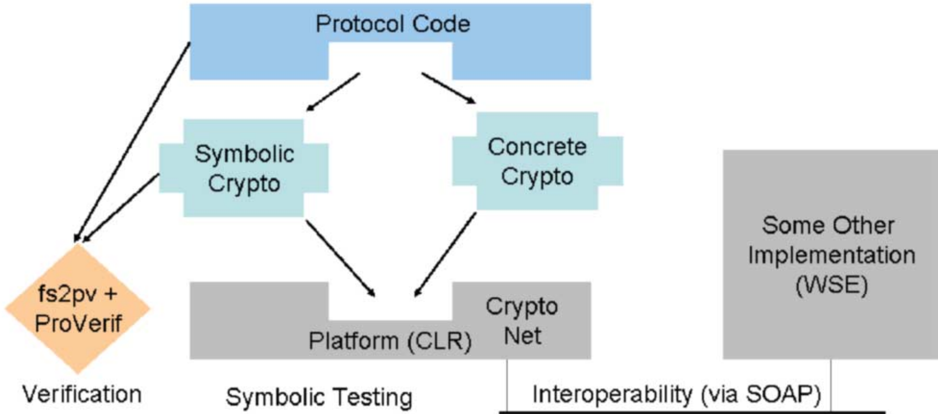
**Keywords.** Security protocols, verification, process calculi

## 1. Introduction

The design and implementation of code involving cryptography remains dangerously difficult. The problem is to verify that an active attacker, possibly with access to some cryptographic keys but unable to guess other secrets, cannot thwart security goals such as authentication and secrecy [35]; it has motivated a serious research effort on the formal analysis of cryptographic protocols, starting with Dolev and Yao [18] and eventually leading to effective verification tools. Hence, it is now feasible to verify abstract models of protocols against demanding threat models.

Still, as with many formal methods, a gap remains between protocol models and their implementations. Distilling a cryptographic model is delicate and time consuming, so that verified protocols tend to be short and to abstract many potentially troublesome details of implementation code. At best, the model and its implementation are related during tedious manual code reviews. Even if, at some point, the model faithfully covers the details of the protocol, it is hard to keep it synchronized with code as it is deployed and used. Hence, despite verification of the abstract model, security flaws may appear in its implementation.

Our thesis is that to verify production code of security protocols against realistic threat models is an achievable research goal. The present paper advances in this direction by contributing a new approach to deriving automatically verifiable models from code. We demonstrate its application, if not to production code, at least to code constitut-




ing a working reference implementation—one suitable for interoperability testing with efficient production systems but itself optimized for clarity not performance.

Our prototype tools analyze cryptographic protocols written in F# [41], a dialect of ML. F# is a good fit for our purposes: it has a simple formal semantics; its datatypes offer a convenient way of programming operations on XML, important for our motivating application area, web services security. Semantically, F# is not so far from languages like Java or C#, and we expect our techniques could be adapted to such languages. We run F# programs on the Common Language Runtime (CLR), and rely on the .NET Framework libraries for networking and cryptographic functions.

The diagram above describes our new language-based approach, which derives verifiable models from executable code. We prefer not to tackle the converse problem, turning a formal model into code, as, though feasible, it amounts to language design and implementation, which generally is harder and takes more engineering effort than model extraction from an existing language. Besides, modern programming environments provide better tool support for writing code than for writing models.

We strive to share most of the code, syntactically and semantically, between the implementation and its model. Our approach is modular, as illustrated by the diagram: we write application code defining protocols against restrictive typed interfaces defining the services exposed by the underlying cryptographic, networking, and other libraries. Further, we write distinct versions of library code only for a few core interfaces, such as those featuring cryptographic algorithms. For example, cryptographic operations are on an abstract type bytes. We provide dual *concrete* and *symbolic* implementations of each operation. For instance, the concrete implementation of bytes is simply as byte arrays, subject to actual cryptographic transforms provided by the .NET Framework. On the other hand, the symbolic implementation defines bytes as algebraic expressions subject to abstract rewriting in the style of Dolev and Yao, and assumed to be a safe abstraction of the concrete implementation.

We formalize the active attacker as an arbitrary program in our source language, able to call interfaces defined by the application code and also the libraries for cryptography and networking. Our verification goals are to show secrecy and authentication properties in the face of all such attackers. Accordingly, we can adapt our threat model by designing suitable interfaces for the benefit of the attacker. The application code implements functions for each role in the protocol, so the attacker can create multiple instances of,

say, initiators and responders, as well as monitor and send network traffic and, in some models, create new principals and compromise some of their credentials.

Given dual implementations for some libraries, we can compile and execute programs both concretely and symbolically. This supports the following tasks:

(1) To obtain a *reference implementation*, we execute application code against concrete libraries. We use the reference implementation for interoperability testing with some other available, black-box implementation. Experimental testing is essential to confirm that the protocol code is functionally correct, and complete for at least a few basic scenarios. (Otherwise, it is surprisingly easy to end up with a model that does not support some problematic features.)
(2) To obtain a *symbolic prototype*, we execute the same application code against symbolic libraries. This allows basic testing and debugging, especially for the expected message formats. Though this guarantees neither wire format interoperability nor any security properties, it is pragmatically useful during the initial stages of code development.
(3) To perform *formal verification*, we run our model extraction tool, called fs2pv, to derive a detailed formal model from the application code and symbolic libraries. Our models are in a variant of the pi calculus [32,1] accepted by ProVerif [15,14]. ProVerif compiles our models to logical clauses and runs a resolution semi-algorithm to prove properties automatically. In case a security property fails, ProVerif can often construct an explicit attack [4].

The fs2pv/ProVerif tool chain is applicable in principle to a broad range of cryptographic protocols, but our motivating examples are those based on the WS-Security [34] standard for securing SOAP [25] messages sent to and from XML web services. WS-Security prescribes how to sign and encrypt parts of SOAP messages. WSE [30] is an implementation of security protocols based on WS-Security. Previous analyses of pi calculus models extracted from WSE by hand have uncovered attacks [9,11], but there has been no previous attempt to check conformance between these models and code automatically. To test the viability of our new approach, we have developed a series of reference implementations of simple web services protocols. They are both tested to be interoperable with WSE and verified via our tool chain. The research challenge in developing these implementations is to confront at once the difficulty of processing standard wire formats, such as WS-Security, and the difficulty of extracting verifiable models from code.

Our model extraction tool, fs2pv, accepts an expressive first-order subset of F# we dub F, with primitives for communications and concurrency. It has a simple formal semantics facilitating model extraction, but disallows higher-order functions and some imperative features. The application code and the symbolic libraries must be within F, but the concrete libraries are in unrestricted F#, with calls to the platform libraries. Formally, we define the attacker to be an arbitrary F program well formed with respect to a restrictive *attacker interface* implemented by the application code. The attacker can only interact with the application code via this interface, which is supplied explicitly to the model extraction tool along with the application code. Although we compile to the pi calculus for verification, the properties proved can be understood independently of the pi calculus. We prove theorems to justify that verification with ProVerif implies properties of source programs defined in terms of F. The principal difficulty in the proofs arises from relating the attacker models at the two levels.

Since security properties within the Dolev-Yao model are undecidable, and we rely on an automatic verifier, there is correct code within F that fails to verify. A cost of our method, then, is that we must adopt a programming discipline within F suitable for automatic verification. For example, we avoid certain uses of recursion. The initial performance results for our prototype tools are encouraging, as much of the performance is determined by the concrete libraries; nonetheless, there is a tension between efficiency of execution and feasibility of verification. To aid the latter, fs2pv chooses between a range of potential semantics for each F function definition (based on abstractions, rewrite rules, relations, and processes).

Our method relies on explicit interfaces describing low-level cryptographic and communication libraries, and on some embedded specifications describing the intended security properties. Model extraction directly analyzes application code using these interfaces plus the code of the symbolic libraries, while ignoring the code of the concrete libraries. Hence, our method can discover bugs in the application code, but not in the trusted concrete libraries.

At present, we have assessed our method only on new code written by ourselves in this style. Many existing protocol implementations rely on well defined interfaces providing cryptographic and other services, so we expect our method will adapt to existing code bases, but this remains future work.

In general, the derivation of security models from code amounts to translating the security-critical parts of the code and safely abstracting the rest. Given an arbitrary program, this task can hardly be automated—some help from the programmer is needed, at least to assert the intended security properties. Further work may discover how to compute safe abstractions directly from the code of concrete libraries. For now, we claim the benefit of symbolic verification of a reference implementation is worth the cost of adding some security assertions in application code and adopting a programming discipline compatible with verification.

In summary, our main contributions are as follows:

(1) An architecture and language semantics to support extraction of verifiable formal models from implementation code of security protocols.
(2) A prototype model extractor fs2pv that translates from F to ProVerif. This tool is one of the first to extract verifiable models from working protocol implementations. Moreover, to the best of our knowledge, it is the first to extract models from code that uses a standard message format (WS-Security) and hence interoperates with other implementations (WSE).
(3) Theorems justifying model extraction: low-level properties proved by ProVerif of a model extracted by fs2pv imply high-level properties expressed in terms of F.
(4) Reference implementations of some typical web services security protocols and mechanisms, both formally verified and tested for interoperability. Our implementation is modular, so that most code is expressed in reusable libraries that give a formal semantics to informal web services security specifications.

Section 2 informally introduces many ideas of the paper in the context of a simple message authentication protocol. Section 3 defines our source language, F, as a subset of F#, and formalizes our desired security properties. Section 4 outlines our techniques for model extraction, and states our main theorems. Section 5 summarizes our experience in writing and verifying code for web services security protocols. Section 6 concludes.

An abridged version [12] of this paper appears in a conference proceedings. A companion report [13] provides additional technical details, including definitions for the target (pi calculus) language, the formal translation, and all proofs.

## 2. A Simple Message Authentication Protocol

We illustrate our method on a very simple, ad hoc protocol example. Section 5 discusses more involved examples.

*The protocol* Our example protocol has two roles, a client that sends a message, and a server that receives it. For the sake of simplicity, we assume that there is only one principal $A$ acting as a client, and only one principal $B$ acting as a server. (Further examples support arbitrarily many principals in each role.)

Our goal here is that the server authenticate the message, even in the presence of an active attacker. To this end, we rely on a password-based message authentication code (MAC). The protocol consists of a single message:

$$\begin{aligned} A \rightarrow B : \text{HMACSHA1}\{\mathit{nonce}\}[\mathit{pwd}_A \mid \mathit{text}] \mid \\ \text{RSAEncrypt}\{\mathit{pk}_B\}[\mathit{nonce}] \mid \mathit{text} \end{aligned}$$

The client acting for principal $A$ sends a single message *text* to the server acting for $B$. The client and server share $A$'s password $pwd_A$, and the client knows $B$'s public key $pk_B$. To authenticate the message *text*, the client uses the one-way keyed hash algorithm HMAC-SHA1 to bind the message with $pwd_A$ and a freshly generated value *nonce*. Since the password is likely to be a weak secret, that is, a secret with low entropy, it may be vulnerable to offline dictionary attacks if the MAC, the message *text*, and the nonce are all known. To protect the password from such guessing attacks, the client encrypts the nonce with $pk_B$.

*Application code* Given interfaces Crypto, Net, and Prins defining cryptographic primitives, communication operations, and access to a database of principal identities, our verifiable application code is a module that implements the following typed interface.

```
pkB: rsa_key
client: str → unit
server: unit → unit
```

The value pkB is the public encryption key for the server. Calling client with a string parameter should send a single message to the server, while calling server creates an instance of the server role that awaits a single message.

In F#, str → unit is the type of functions from the type str, which is an abstract type of strings defined by the Crypto interface, to the empty tuple type unit. The Crypto interface also provides the abstract type rsa_key of RSA keys.

The exported functions client and server rely on the following functions to manipulate messages.

```
let mac nonce password text =
  Crypto.hmacsha1 nonce
    (concat (utf8 password) (utf8 text))
```

```
let make text pk password =
  let nonce = mkNonce() in
  (mac nonce password text,
   Crypto.rsa_encrypt pk nonce, text)

let verify (m,en,text) sk password =
  let nonce = Crypto.rsa_decrypt sk en in
  if not (m = mac nonce password text)
  then failwith "bad MAC"
```

The first function, mac, takes three arguments—a nonce, a shared password, and the message text—and computes their joint cryptographic hash using some implementation of the HMAC-SHA1 algorithm provided by the cryptographic library. As usual in dialects of ML, types may be left implicit in code, but they are nonetheless verified by the compiler; mac has type bytes $\rightarrow$ str $\rightarrow$ str $\rightarrow$ bytes. The functions concat and utf8 provided by Crypto perform concatenation of byte arrays and an encoding of strings into byte arrays.

The two other functions define message processing, for senders and receivers, respectively. Function make creates a message: it generates a fresh nonce, computes the MAC, and also encrypts the nonce under the public key pk of the intended receiver, using the rsa_encrypt algorithm. The resulting message is a triple comprising the MAC, the encrypted nonce, and the text. Function verify performs the converse steps: it decrypts the nonce using the private key skd, recomputes the MAC and, if the resulting value differs from the received MAC, throws an exception (using the failwith primitive).

Although fairly high-level, our code includes enough details to be executable, such as the details of particular algorithms, and the necessary utf8 conversions from strings (for password and text) to byte arrays.

In the following code defining protocol roles, we rely on events to express intended security properties. Events roughly correspond to assertions used for debugging purposes, and they have no effect on the program execution. Here, we define two kinds of events, Send(text) to mark the intent to send a message with content text, and Accept(text) to mark the acceptance of text as genuine. Accordingly, client uses a primitive function log to log an event of the first kind before sending the message, and server logs an event of the second kind after verifying the message. Hence, if our protocol is correct, we expect every Accept(text) event to be preceded by a matching Send(text) event. Such a correspondence between events is a common way of specifying authentication.

The client code relies on the network address of the server, the shared password, and the server's public key:

```
let address = S "http://server.com/pwdmac"
let pwdA = Prins.getPassword(S "A")
let pkB = Prins.getPublicKey(S "B")

type Ev = Send of str | Accept of str
```

```
let client text =
  log(Send(text));
  Net.send address (marshall (make text pkB pwdA))
```

Here, the function getPassword retrieves *A*'s password from the password database, and getPublicKey extracts *B*'s public key from the local X.509 certificate database. The function S is defined by Crypto; the expression S `"A"`, for example, is an abstract string representing the literal `"A"`. The function client then runs the protocol for sending text; it builds the message, then uses Net.send, a networking function that posts the message as an HTTP request to address.

Symmetrically, the function server attempts to receive a single message by accepting a message and verifying its content, using *B*'s private key for decryption.

```
let skB = Prins.getPrivateKey(S "B")
let server () =
  let m,en,text = unmarshall (Net.accept address) in
  verify (m,en,text) skB pwdA; log(Accept(text))
```

The functions marshall and unmarshall serialize and deserialize the message triple—the MAC, the encrypted nonce, and the text—as a string, used here as a simple wire format. (We present an example of the resulting message below.) These functions are also part of the verified application code; we omit their details.

*Concrete and symbolic libraries* The application code listed above makes use of a Crypto library for cryptographic operations, a Net library for network operations, and a Prins library offering access to a principal database. The concrete implementations of these libraries are F# modules containing functions that are wrappers around the corresponding platform (.NET) cryptographic and network operations.

To obtain a complete symbolic model of the program, we also develop symbolic implementations of these libraries as F# modules with the same interfaces. These symbolic libraries are within the restricted subset F we define in the next section, and rely on a small module Pi defining name creation, channel-based communication, and concurrency in the style of the pi calculus. Functions Pi.send and Pi.recv allow message passing on channels, functions Pi.name and Pi.chan generate fresh names and channels, and a function Pi.fork runs its function argument in parallel. The members of Pi are primitive in the semantics of F. The Pi module is called from the symbolic libraries during symbolic evaluation and formal verification; it is not called directly from application code and plays no part in the concrete implementation.

The listings above show the two implementations of the Crypto interface. The concrete implementation defines bytes as primitive arrays of bytes, and essentially forwards all calls to standard cryptographic libraries of the .NET platform. In contrast, the symbolic implementation defines bytes as an algebraic datatype, with symbolic constructors and pattern matching for representing cryptographic primitives. This internal representation is accessible only in this library implementation. For instance, hmacsha1 is implemented as a function that builds an HmacSha1(k,x) term; since no inverse function is provided, this abstractly defines a perfect, collision-free one-way function. More interestingly, RSA public key encryptions are represented by RsaEncrypt terms, decomposed

```
module Crypto // concrete code in F#
open System.Security.Cryptography
type bytes = byte[]
type rsa_key = RSA of RSAParameters
...
let rng = new RNGCryptoServiceProvider ()
let mkNonce () =
  let x = Bytearray.make 16 in
  rng.GetBytes x; x
...
let hmacsha1 k x =
  new HMACSHA1(k).ComputeHash x
...
let rsa = new RSACryptoServiceProvider()
let rsa_keygen () = ...
let rsa_pub (RSA r) = ...
let rsa_encrypt (RSA r) (v:bytes) = ...
let rsa_decrypt (RSA r) (v:bytes) =
  rsa.ImportParameters(r);
  rsa.Decrypt(v,false)
```

```
module Crypto // symbolic code in F
type bytes =
  | Name of Pi.name
  | HmacSha1 of bytes * bytes
  | RsaKey of rsa_key
  | RsaEncrypt of rsa_key * bytes
  ...
and rsa_key = PK of bytes | SK of bytes
...
let freshbytes label = Name (Pi.name label)
let mkNonce () = freshbytes "nonce"
...
let hmacsha1 k x = HmacSha1(k,x)
...
let rsa_keygen () = SK (freshbytes "rsa")
let rsa_pub (SK(s)) = PK(s)
let rsa_encrypt s t = RsaEncrypt(s,t)
let rsa_decrypt (SK(s)) e = match e with
  | RsaEncrypt(pke,t) when pke = PK(s) → t
  | _ → failwith "rsa_decrypt failed"
```

only by a function rsa_decrypt that can verify that the valid decryption key is provided along with the encrypted term.

Similarly, the concrete implementation of Net contains functions, such as send and accept, that call into the platform's HTTP library (System.Net.WebRequest), whereas the symbolic implementation of these functions simply enqueues and dequeues messages from a shared buffer implemented with the Pi module as a channel. We outline the symbolic implementation of Net below.

```
module Net // symbolic code in F
...
let httpchan = Pi.chan()
let send address msg =
  Pi.send httpchan (address,msg)
let accept address =
  let (addr,msg) = Pi.recv httpchan in
  if addr = address then msg else ...
```

The function send adds a message to the channel httpchan and the function accept removes a message from the channel.

In this introductory example, we have a fixed population of two principals, so the values for $A$'s password and $B$'s key pair can simply be retrieved from the third interface Prins: the concrete implementation of Prins binds them to constants; its symbolic implementation binds them to fixed names generated by calling Pi.name. In general, a concrete implementation would retrieve keys from the operating system key store, or prompt the user for a password. The symbolic version implements a database of passwords and keys using a channel kept hidden from the attacker.

Next, we describe how to build both a concrete reference implementation and a symbolic prototype, in the sense of Section 1.

*Concrete execution* To test that the protocol runs correctly, we run the F# compiler on the F application code, the concrete F# implementations of Crypto, Net, and Prins, together with the following top-level F# code to obtain a single executable, say run. Depending on its command line argument, this executable runs in client or server mode:

```
do match Sys.argv.(1) with
  | "client" → client (S Sys.argv.(2))
  | "server" → server ()
  | _ → printf "Usage: run client txt\n";
      printf " or: run server\n"
```

The library function call Sys.argv.($n$) returns the $n$th argument on the command line. As an example, we can execute the command run client Hi on some machine, execute run server on some other machine that listens on address, and observe the protocol run to completion. This run of the protocol involves our concrete implementation of (HTTP-based) communications sending and receiving the encoded string "FADCIzZhW3XmgUABgRJj1KjnWy...".

*Symbolic execution* To experiment with the protocol code symbolically, we run the F# compiler on the F application code, the symbolic F implementations of Crypto, Net, and Prins, and the F# implementation of the Pi interface, together with the following top-level F code, that conveniently runs instances of the client and of the server within a single executable.

```
do Pi.fork (fun()→ client (S "Hi"))
do server ()
```

The communicated message prints as follows

```
HMACSHA1{nonce3}[pwd1 | 'Hi'] |
  RSAEncrypt{PK(rsa_secret2)}[nonce3] | 'Hi'
```

where pwd1, rsa_secret2, and nonce3 are the symbolic names freshly generated by the Pi module. This message trace reveals the structure of the abstract byte arrays in the communicated message, and hence is more useful for debugging than the concrete message trace. We have found it useful to test application code by symbolic execution (and even symbolic debugging) before testing them concretely on a network.

*Modelling the opponent* We introduce our language-based threat model for protocols developed in F. (Section 3 describes the formal details.)

Let $S$ be the F program that consists of the application code plus the symbolic libraries. The program $S$, which largely consists of code shared with the concrete implementation, constitutes our formal model of the protocol.

Let $O$ be any F program that is well formed with respect to the interface exported by the application code (in this case, the value pkB and the functions client and server), plus the interfaces Crypto and Net. By well formed, we mean that $O$ only uses external values

and calls external functions explicitly listed in these interfaces. Moreover, $O$ can call all the operations in the Pi interface, as these are primitives available to all F programs. We take the program $O$ to represent a potential attacker on the formal model $S$ of the protocol, a counterpart to an active attacker on a concrete implementation. (Treating an attacker as an arbitrary F program develops the idea of an attacker being an arbitrary parallel process, as in the spi calculus [2].)

Giving $O$ access to the Crypto and Net interfaces, but not Prins, corresponds to the Dolev-Yao [18] model of an attacker able to perform symbolic cryptography, and monitor and send network traffic, but unable to access principals' credentials directly. In particular, Net.send enables the attacker to send any message to the server while Net.accept enables the attacker to intercept any message sent to the server. The functions Crypto.rsa_encrypt and Crypto.rsa_decrypt enable encryption and decryption with keys known to the attacker; Crypto.rsa_keygen and Crypto.mkNonce enable the generation of fresh keys and nonces; Crypto.hmacsha1 enables MAC computation.

Giving $O$ access to client and server allows it to create arbitrarily many instances of protocol roles, while access to pkB lets $O$ encrypt messages for the server. (We can enrich the interface to give the opponent access to the secret credentials of some principals, and to allow the generation of arbitrarily many principal identities.) Since pwdA, skB, and log are not included in the attacker interface, the attacker has no direct access to the protocol secrets and cannot log events directly.

Formal verification aims to establish secrecy and authentication properties for all programs $S$ $O$ assembled from the given system $S$ and any attacker program $O$.

In particular, the message authentication property of our example protocol is expressed as correspondences [42] between events logged by code within $S$. For all $O$, we want that in every run of $S$ $O$, every Accept event is preceded by a corresponding Send event. In our syntax (based on that of ProVerif), we express this correspondence assertion as:

**ev**:Accept(x) $\Rightarrow$ **ev**:Send(x)

*Formal verification* We can check correspondences at runtime during any particular symbolic run of the program; the more ambitious goal of formal verification is to prove them for all possible runs and attackers. To do so, we run our model extractor fs2pv on the F application code, the symbolic F implementations of Crypto, Net, and Prins, and the attacker interface as described above. The result is a pi calculus script with embedded correspondence assertions suitable for verification with ProVerif. In the simplest case, F functions compile to pi calculus processes, while the attacker interface determines which names are published to the pi calculus attacker. For our protocol, ProVerif immediately succeeds.

Conversely, consider for instance a variant of the protocol where the MAC computation does not actually depend on the text of the message—essentially transforming the MAC into a session cookie:

```
let mac nonce password text = hmacsha1 nonce
  (concat (utf8 password) (utf8 (S "cookie")))
```

For the resulting script, ProVerif automatically finds and reports an active attack, whereby the attacker intercepts the client message and substitutes any text for the client's

text in the message. Experimentally, we can confirm the attack found in the analysis, by writing in F an instance of the attacker program $O$ that exploits our interface. Here, the attack may be written:

```
do fork(fun()→ client (S "Hi"));
   let (nonce, mac, _) = unmarshall (Net.accept address) in
   fork(fun()→ server());
   Net.send address (marshall (nonce, mac, S "Foo"))
```

This code first starts an instance of the client, intercepts its message, starts an instance of the server, and forwards an amended message to it. Experimentally, we observe that the attack succeeds, both concretely and symbolically. At the end of those runs, two events Send "Hi" and Accept "Foo" have been emitted, and our authentication query fails. Once the attack is identified and the protocol corrected, this attacker code may be added to the test suite for the protocol.

In addition to authentication, we verify secrecy properties for our example protocol. Via ProVerif [15], we can query whether a protocol allows an attacker to guess a weak secret and then verify the guess—if so, the attacker can mount an offline guessing attack. In the case of our protocol, ProVerif shows the password is protected against offline guessing attacks. Conversely, if we consider a variant of the protocol that passes the nonce in the clear, we find an attack that can also be written as a concrete F program.

## 3. Formalizing a Subset of F#

This section defines the untyped subset F of F# in which we write application code and symbolic libraries. We specify the syntax of F, describe its informal and formal semantics, and define security properties.

The language F consists of: a first-order functional core; algebraic datatypes with pattern-matching (such as the type bytes in the symbolic implementation of Crypto); a few concurrency primitives in the style of the pi calculus; and a simple type-free module system with which we formalize the attacker model introduced in the previous section. (Although we do not rely on type safety in the formal definition, F programs can be typechecked by the F# compiler.)

*Syntax and Informal Semantics of F* In the syntax below, $\ell$ ranges over first-order functions (such as freshBytes or hmacsha1 in Crypto) and $f$ ranges over datatype constructors (such as Name or Hmacsha1 in the type bytes in Crypto). Functions and constructors are either primitive, or introduced by function or datatype declarations. The primitives include the communication functions Pi.send, Pi.recv, and Pi.name described in the previous section. The concurrency operator Pi.fork is a higher-order function; we build Pi.fork into the syntax of F. In F, we treat Pi.chan as a synonym for Pi.name; they have different types but both create fresh atomic names. We omit the “Pi.” prefix for brevity.

**Syntax of F:**

| | |
|---|---|
| $x, y, z$ | variable |
| $a, b$ | name |
| $f$ | constructor (uncurried) |

| | |
|---|---|
| $\ell$ | function (curried) |
| $\text{true}, \text{false}, \text{tuple}n \quad n \geq 0$ | primitive constructors |
| $\text{name}, \text{send}, \text{recv}, \text{log}, \text{failwith}$ | primitive functions |
| $M, N ::=$ | value |
| $\quad x$ | variable |
| $\quad a$ | name |
| $\quad f(M_1, \ldots, M_n)$ | constructor application |
| $e ::=$ | expression |
| $\quad M$ | value |
| $\quad \ell\, M_1 \ \ldots \ M_n$ | function application |
| $\quad \text{fork}(\mathbf{fun}() \rightarrow e)$ | fork a parallel thread |
| $\quad \mathbf{match}\ M\ \mathbf{with}(\mid M_i \rightarrow e_i)^{i \in 1..n}$ | pattern match |
| $\quad \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ | sequential evaluation |
| $d ::=$ | declaration |
| $\quad \mathbf{type}\ s = (\mid f_i\ \mathbf{of}\ s_{i1} * \ldots * s_{im_i})^{i \in 1..n}$ | datatype declaration |
| $\quad \mathbf{let}\ x\ = e$ | value declaration |
| $\quad \mathbf{let}\ \ell\, x_1 \ldots x_n = e \quad n > 0$ | function declaration |
| $S ::= d_1 \cdots d_n$ | system: list of declarations |

We rely on the following syntactic conventions. For any phrase of syntax $\phi$, we write $fv(\phi)$ and $fn(\phi)$ for the sets of variables and names occurring free in $\phi$. To facilitate the translation from F to the pi calculus, we assume each function $\ell$ is a pi calculus name, so that, for example, $fn(\ell\, M_1 \ \ldots \ M_n) = \{\ell\} \cup fn(M_1) \cup \cdots \cup fn(M_n)$. A phrase of syntax $\phi$ is *closed* iff $fv(\phi) = \varnothing$. We identify phrases of syntax up to consistent renaming of bound variables and names; that is, $\phi = \phi'$ means that $\phi$ and $\phi'$ are the same up to such renaming. Let $\sigma$ range over ground substitutions $\{M_1/x_1, \ldots, M_n/x_n\}$ of values for variables, where $fv(M_i) = \varnothing$.

A system $S$ is a sequence of declarations. We write the list $S$ as $\varnothing$ when it is empty. A datatype declaration introduces a new type and its constructors (much like a union type with tags in C); the type expressions $s$, $s_{ij}$ are ignored in F. A value declaration $\mathbf{let}\ x\ = e$ triggers the evaluation of expression $e$ and binds the result to $x$. A function declaration $\mathbf{let}\ \ell\, x_1 \ldots x_n = e$ defines function $\ell$ with formal parameters $x_1 \ldots x_n$ and function body $e$. These functions may be recursive.

A value $M$ is a variable, a name, or a constructor application. Names model channels, keys, and nonces. Names can only be introduced during evaluation by calling the primitive name. Source programs contain no free names. Expressions denote potentially concurrent computations that return values. Primitive functions mostly represent communication and concurrency: name() returns a freshly generated name; send $M\ N$ sends $N$ on the channel $M$; recv $M$ returns the next value received on channel $M$; log $M$ logs the event $M$; failwith $M$ represents a thrown exception; and $\text{fork}(\mathbf{fun}() \rightarrow e)$ evaluates $e$ in parallel. (We need not model exception handling in F as we rely on exceptions only to represent fatal errors.) If $\ell$ has a declaration, the application $\ell\, M_1 \ \ldots \ M_n$ invokes the body of the declaration with actual parameters $M_1, \ldots, M_n$. A $\mathbf{match}\ M\ \mathbf{with}(\mid M_i \rightarrow e_i)^{i \in 1..n}$ runs $e_i$ for the least $i$ such that pattern $M_i$ matches the value $M$; if the pattern $M_i$ contains variables, they are bound in $e_i$ by matching with $M$. If there are two or more occurrences of a variable in a pattern, matching must bind each to the same value. (Strictly speaking, F# forbids patterns with multiple occurrences of the same variable. Still, the effect of

any such pattern in F can be had in F# by renaming all but one of the occurrences and adding one or more equality constraints via a **when** clause.) Finally, **let** $x = e_1$ **in** $e_2$ first evaluates $e_1$ to a value $M$, then evaluates $e_2\{M/x\}$, that is, the outcome of substituting $M$ for each free occurrence of $x$ in $e_2$.

In addition to the core syntax of F, we recover useful syntax supported by F# as follows. The first three rules allow expressions to be written in places where only values are allowed by the core syntax; these rules only apply when the left-hand side is not within the core syntax.

**Derived Expressions:**

$f(e_1,\dots,e_n) \triangleq \textbf{let } x_1 = e_1 \textbf{ in} \dots \textbf{let } x_n = e_n \textbf{ in } f(x_1,\dots,x_n) \quad x_i \text{ fresh}$
$\ell\ e_1\ \dots\ e_n \triangleq \textbf{let } x_1 = e_1 \textbf{ in} \dots \textbf{let } x_n = e_n \textbf{ in } \ell\ x_1\ \dots\ x_n \quad x_i \text{ fresh}$
$\textbf{match } e_0 \textbf{ with}(|\ M_i \rightarrow e_i)^{i \in 1..n} \triangleq \textbf{let } x_0 = e_0 \textbf{ in match } x_0 \textbf{ with}(|\ M_i \rightarrow e_i)^{i \in 1..n} \quad x_0 \text{ fresh}$

$f \triangleq f() \quad \text{where constructor } f \text{ has arity } 0$
$(e_1,\dots,e_n) \triangleq \text{tuple}n(e_1,\dots,e_n) \qquad \text{where } n \geq 0$
$\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \triangleq \textbf{match } e \textbf{ with } |\ \text{true} \rightarrow e_1 \ |\ \text{false} \rightarrow e_2$
$e_1 = e_2 \triangleq \textbf{match } (e_1,e_2) \textbf{ with } |\ (x,x) \rightarrow \text{true} \ |\ (x,y) \rightarrow \text{false}$
$e_1; e_2 \triangleq \textbf{let } x = e_1 \textbf{ in } e_2 \qquad \text{where } x \notin fv(e_2)$

*Operational Semantics of F* Next, we formalize the operational semantics of F and the idea of safety with respect to a query. Let a *configuration*, $C$, be a multiset of running systems and logged events. We write $C \mid C'$ for the composition of configurations $C$ and $C'$. To formalize that configurations are multisets, we identify configurations up to a *structural equivalence* relation, $C \equiv C'$, that includes laws of associativity and commutativity for composition. It also includes a law $C \mid \varnothing \equiv C$ to allow deletion of an empty sequence of declarations, $\varnothing$.

**Syntax of F Configurations, and Structural Equivalence:**

$C ::= S \mid \textbf{event } M \mid (C \mid C')$

$C_2 \equiv C_1 \Rightarrow C_1 \equiv C_2 \qquad C_1 \mid C_2 \equiv C_2 \mid C_1$
$C_1 \equiv C_2, C_2 \equiv C_3 \Rightarrow C_1 \equiv C_3 \qquad C_1 \mid (C_2 \mid C_3) \equiv (C_1 \mid C_2) \mid C_3$
$C_1 \equiv C_2 \Rightarrow C_1 \mid C \equiv C_2 \mid C \qquad C \mid \varnothing \equiv C$

The following rules define a small-step reduction semantics on configurations.

**Reduction Rules: $C \rightarrow C'$ where $C$ and $C'$ are closed**

$C_1 \rightarrow C_2 \text{ if } C_1 \equiv C_1', C_1' \rightarrow C_2', C_2' \equiv C_2$
$C_0 \mid d\ \ S \rightarrow C_0 \mid S \quad \text{if } d \text{ is a datatype declaration}$
$C_0 \mid d\ \ S \rightarrow C_0 \mid d \mid S \quad \text{if } d \text{ is a function declaration, } S \neq \varnothing$

$C_0 \mid \textbf{let } x = M\ \ S \rightarrow C_0 \mid S\{M/x\}$
$C_0 \mid \textbf{let } x = \ell\ M_1\ \dots\ M_n\ \ S \rightarrow C_0 \mid \textbf{let } x = e\{M_1/x_1,\dots,M_n/x_n\}\ \ S$
$\quad \text{if } C_0 = C_1 \mid \textbf{let } \ell\ x_1 \dots x_n = e$
$C_0 \mid \textbf{let } x = \text{name } ()\ \ S \rightarrow C_0 \mid S\{a/x\} \quad \text{if } a \notin fn(C_0,S)$

$C_0 \mid \textbf{let}\ x_1 = \text{send}\ M\ N\ \ S_1 \mid \textbf{let}\ x_2 = \text{recv}\ M\ \ S_2 \rightarrow C_0 \mid S_1\{()/x_1\} \mid S_2\{N/x_2\}$
$C_0 \mid \textbf{let}\ x = \log M\ \ S \rightarrow C_0 \mid \textbf{event}\ M \mid S\{()/x\}$
$C_0 \mid \textbf{let}\ x = \text{fork}(\textbf{fun}() \rightarrow e)\ \ S \rightarrow C_0 \mid \textbf{let}\ x = e \mid S\{()/x\}$
$C_0 \mid \textbf{let}\ x = \textbf{match}\ M\ \textbf{with}\ (\mid M_i \rightarrow e_i)^{i \in 1..n}\ \ S \rightarrow C_0 \mid \textbf{let}\ x = e_1\sigma\ \ S \quad \text{if } M = M_1\sigma$
$C_0 \mid \textbf{let}\ x = \textbf{match}\ M\ \textbf{with}\ (\mid M_i \rightarrow e_i)^{i \in 1..n}\ \ S$
$\quad \rightarrow C_0 \mid \textbf{let}\ x = \textbf{match}\ M\ \textbf{with}\ (\mid M_i \rightarrow e_i)^{i \in 2..n}\ \ S \quad \text{if } \neg\exists\sigma.\ M = M_1\sigma$
$C_0 \mid \textbf{let}\ x = (\textbf{let}\ y = e_1\ \textbf{in}\ e_2)\ \ S \rightarrow C_0 \mid \textbf{let}\ y = e_1\ \textbf{let}\ x = e_2\ \ S \quad y \notin fv(S)$

The first rule allows configurations to be rearranged up to $C \equiv C'$ when calculating a reduction. The second simply discards a top-level datatype declaration in a system; types have no effect at runtime. The third forks a top-level function declaration $d$ as a separate system consisting just of $d$; this system is itself insert, but it can be called from other systems running in parallel. The remaining rules apply to a top-level value declaration **let** $x = e$, for some $e$, running in a context including a configuration $C_0$, and specify how the expression $e$ evaluates in that context. These rules formalize the description of expression evaluation given earlier in this section.

The only primitive function not to appear in a reduction rule is failwith; applications of the form failwith $M$ are simply stuck (although in F# they raise an exception).

*A Simple Example* We consider an example system $S_{10}$ representing transmission of a single encrypted message from an initiator to a responder. The system $S_{10}$ consists of a sequence of ten declarations, which we define below.

$$S_{10} \triangleq d_{\text{Ev}}\ d_{\text{Cipher}}\ d_{\text{enc}}\ d_{\text{dec}}\ d_{\text{net}}\ d_{\text{key}}\ d_{\text{init}}\ d_{\text{resp}}\ d_{\text{u1}}\ d_{\text{u2}}$$

The first two declarations are of types: a type of events (as in Section 2) and a type of symmetric-key authenticated encryptions (a much simplified version of the type bytes from Section 2).

$$d_{\text{Ev}} \triangleq \textbf{type}\ \text{Ev = Send}\ \textbf{of}\ \text{string} \mid \text{Accept}\ \textbf{of}\ \text{string}$$
$$d_{\text{Cipher}} \triangleq \textbf{type}\ \text{Cipher = Enc}\ \textbf{of}\ \text{string} * \text{string}$$

Next, we declare an encryption function enc and a decryption function dec. (The latter includes a pattern (Enc(p,z),z) containing two occurrences of the same variable. As mentioned above, such patterns are allowed in F but not literally in F#, although we can achieve the same effect in F# by writing (Enc(p,z),z') **when** z=z'.)

$$d_{\text{enc}} \triangleq \textbf{let}\ \text{enc x y = Enc(x,y)}$$
$$d_{\text{dec}} \triangleq \textbf{let}\ \text{dec x y =}\ \textbf{match}\ \text{(x,y)}\ \textbf{with} \mid \text{(Enc(p,z),z)} \rightarrow \text{p}$$

The next four declarations generate names for a shared network channel (net) intended to be public, and a shared symmetric key (key) intended to be known only to the initiator and responder, and define the initiator and responder role as functions init and resp. The initiator logs a Send event, creates an encryption, and sends it on the network channel. The responder receives a message, decrypts it, and, if the decryption succeeds, logs an Accept event.

$$d_{\text{net}} \triangleq \textbf{let}\ \text{net} = \text{name}()$$

$$d_{\text{key}} \triangleq \textbf{let}\ \text{key} = \text{name}()$$

$$d_{\text{init}} \triangleq \textbf{let}\ \text{init x} = \text{log (Send(x))};\ \textbf{let}\ \text{c} = \text{enc x key}\ \textbf{in}\ \text{send net c}$$

$$d_{\text{resp}} \triangleq \textbf{let}\ \text{resp ()} = \textbf{let}\ \text{m} = \text{recv net}\ \textbf{in}\ \textbf{let}\ \text{x} = \text{dec m key}\ \textbf{in}\ \text{log (Accept(x))}$$

The final two declarations simply fork a single instance of the initiator role and a single instance of the responder role.

$$d_{\text{u1}} \triangleq \textbf{let}\ \text{u1} = \text{fork}(\textbf{fun}() \rightarrow \text{init}\ \texttt{"msg1"})$$

$$d_{\text{u2}} \triangleq \textbf{let}\ \text{u2} = \text{fork}(\textbf{fun}() \rightarrow \text{resp}\ ())$$

To illustrate the rules of the formal semantics, we calculate a reduction sequence in which an encryption of `"msg1"` flows from the initiator to the responder. We eliminate empty systems with the equation $C \mid \varnothing \equiv C$. We begin the calculation with the following steps: the two type declarations are discarded, and the first two function declarations are forked as separate systems.

$$\begin{aligned} S_{10} &\rightarrow d_{\text{Cipher}}\ d_{\text{enc}}\ d_{\text{dec}}\ d_{\text{net}}\ d_{\text{key}}\ d_{\text{init}}\ d_{\text{resp}}\ d_{\text{u1}}\ d_{\text{u2}} \\ &\rightarrow d_{\text{enc}}\ d_{\text{dec}}\ d_{\text{net}}\ d_{\text{key}}\ d_{\text{init}}\ d_{\text{resp}}\ d_{\text{u1}}\ d_{\text{u2}} \\ &\rightarrow d_{\text{enc}} \mid d_{\text{dec}}\ d_{\text{net}}\ d_{\text{key}}\ d_{\text{init}}\ d_{\text{resp}}\ d_{\text{u1}}\ d_{\text{u2}} \\ &\rightarrow d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{net}}\ d_{\text{key}}\ d_{\text{init}}\ d_{\text{resp}}\ d_{\text{u1}}\ d_{\text{u2}} \end{aligned}$$

The next part of the computation generates fresh, distinct names n and k and binds them to the variables net and key, respectively. The following abbreviations record the outcome of substituting these names for the variables in init and resp.

$$d_{\text{init}}^{\text{n}} \triangleq d_{\text{init}}\{\text{n}/\text{net}\} \qquad d_{\text{init}}^{\text{n k}} \triangleq d_{\text{init}}^{\text{n}}\{\text{k}/\text{key}\}$$

$$d_{\text{resp}}^{\text{n}} \triangleq d_{\text{resp}}\{\text{n}/\text{net}\} \qquad d_{\text{resp}}^{\text{n k}} \triangleq d_{\text{resp}}^{\text{n}}\{\text{k}/\text{key}\}$$

We have the following reductions in which n and k are generated, and the initiator and responder functions are forked as separate systems.

$$\begin{aligned} & d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{net}}\ d_{\text{key}}\ d_{\text{init}}\ d_{\text{resp}}\ d_{\text{u1}}\ d_{\text{u2}} \\ &\rightarrow d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{key}}\ d_{\text{init}}^{\text{n}}\ d_{\text{resp}}^{\text{n}}\ d_{\text{u1}}\ d_{\text{u2}} \\ &\rightarrow d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{init}}^{\text{n k}}\ d_{\text{resp}}^{\text{n k}}\ d_{\text{u1}}\ d_{\text{u2}} \\ &\rightarrow d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{init}}^{\text{n k}} \mid d_{\text{resp}}^{\text{n k}}\ d_{\text{u1}}\ d_{\text{u2}} \\ &\rightarrow d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{init}}^{\text{n k}} \mid d_{\text{resp}}^{\text{n k}} \mid d_{\text{u1}}\ d_{\text{u2}} \end{aligned}$$

In the next segment of the computation, we fork instances of the initiator and responder as separate threads. As a shorthand, let $C_0 = d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{init}}^{\text{n k}} \mid d_{\text{resp}}^{\text{n k}}$.

$d_{\text{enc}} \mid d_{\text{dec}} \mid d^{\text{n k}}_{\text{init}} \mid d^{\text{n k}}_{\text{resp}} \mid d_{\text{u1}}\ d_{\text{u2}}$
$= C_0$ | **let** u1 = fork(**fun**() → init "msg1") **let** u2 = fork(**fun**() → resp ())
$\rightarrow C_0$ | **let** u1 = init "msg1" | **let** u2 = fork(**fun**() → resp ())
$\rightarrow C_0$ | **let** u1 = init "msg1" | **let** u2 = resp ()

The initiator logs a Send event and prepares to send the encrypted message on the channel $n$. Let $C_1 = C_0$ | **let** u2 = resp ().

$C_0$ | **let** u1 = init "msg1" | **let** u2 = resp ()
$\rightarrow C_1$ | **let** u1 = (log (Send("msg1")); **let** c=enc "msg1"k **in** send n c)
$\rightarrow C_1$ | **let** u3 = log (Send("msg1")) **let** u1=(**let** c=enc "msg1"k **in** send n c)
$\rightarrow C_1$ | **event** Send("msg1") | **let** u1=(**let** c=enc "msg1"k **in** send n c)
$\rightarrow C_1$ | **event** Send("msg1") | **let** c=enc "msg1"k **let** u1=send n c
$\rightarrow C_1$ | **event** Send("msg1") | **let** c=Enc("msg1",k) **let** u1=send n c
$\rightarrow C_1$ | **event** Send("msg1") | **let** u1=send n (Enc("msg1",k))

Next, we consider reductions of the responder **let** u2 = resp (). In fact, it could have reduced in parallel with some of the reductions shown above; we are not here attempting to show all possible interleavings. As a further abbreviation, let $C_2 = C_0$ | **event** Send("msg1") | **let** u1=send n (Enc("msg1",k)).

$C_1$ | **event** Send("msg1") | **let** u1=send n (Enc("msg1",k))
$= C_2$ | **let** u2 = resp ()
$\rightarrow C_2$ | **let** u2 = (**let** m = recv n **in let** x = dec m k **in** log (Accept(x)))
$\rightarrow C_2$ | **let** m = recv n **let** u2 = (**let** x = dec m k **in** log (Accept(x)))

At this point, the encrypted message can pass between the sender and the receiver. We end the calculation with the following steps. Let $C_3 = C_0$ | **event** Send("msg1").

$C_2$ | **let** m = recv n **let** u2 = (**let** x = dec m k **in** log (Accept(x)))
$= C_3$ | **let** u2 = (**let** x = dec (Enc("msg1",k)) k **in** log (Accept(x)))
$\rightarrow C_3$ | **let** x = dec (Enc("msg1",k)) k **let** u2 = log (Accept(x)))
$\rightarrow C_3$ | **let** x = **match** (Enc("msg1",k),k) **with** | (Enc(p,z),z) → p
**let** u2 = log (Accept(x)))
$\rightarrow C_3$ | **let** x = "msg1" **let** u2 = log (Accept(x)))
$\rightarrow C_3$ | **let** u2 = log (Accept("msg1")))
$\rightarrow C_3$ | **event** Accept("msg1")

In summary, we have calculated the following sequence of reductions.

$S_{10} \rightarrow^{+} d_{\text{end}} \mid_{\text{dec}} \mid d^{\text{n k}}_{\text{init}} \mid d^{\text{n k}}_{\text{resp}}$ | **event** Send("msg1") | **event** Accept("msg1")

*Formation Judgments for Expressions and Systems* We use system interfaces to control the capabilities of the opponent. An *interface*, $I$, records the set of values, constructors, and functions imported or exported by a system. Since our verification method does not depend on types, F interfaces omit type structure and track only the distinction between values, constructors, and functions, plus the arity of constructors and functions.

**Interfaces:**

| | |
|---|---|
| $\mu ::= x{:}\mathbf{val} \mid f{:}\mathbf{ctor}\ n \mid \ell{:}\mathbf{fun}\ n$ | mention: value, constructor, or function |
| $I ::= \mu_1, \ldots, \mu_n$ | interface (unordered sequence) |

For example, let Prim be the following interface, which describes the F primitives, where $m$ is an arbitrary maximum width of tuples.

true: **ctor** 0, false: **ctor** 0, (tuple$i$: **ctor** $i)^{i \in 1..m}$,
failwith: **fun** 1, log: **fun** 1, Pi.name: **fun** 1, Pi.chan: **fun** 1,
Pi.send: **fun** 2, Pi.recv: **fun** 1, Pi.fork: **fun** 1

As another example, let $I_{pub}$ be the following interface, which enumerates the functions exported by the symbolic libraries together with the application code for the example protocol in Section 2.

Net.send: **fun** 2, Net.accept: **fun** 1,
Crypto.S: **fun** 1, Crypto.iS: **fun** 1,
Crypto.base64: **fun** 1, Crypto.ibase64: **fun** 1,
Crypto.utf8: **fun** 1, Crypto.iutf8: **fun** 1,
Crypto.concat: **fun** 2, Crypto.iconcat: **fun** 1,
Crypto.mkNonce: **fun** 1, Crypto.mkPassword: **fun** 1,
Crypto.rsa_keygen: **fun** 1, Crypto.rsa_pub: **fun** 1,
Crypto.rsa_encrypt: **fun** 2, Crypto.rsa_decrypt: **fun** 2,
Crypto.hmacsha1: **fun** 2,
pkB: **val**, client: **fun** 1, server: **fun** 1

To define when a system exports an interface, we introduce inductively-defined *formation judgments* for expressions and systems. Let $dom(I)$ be the set of variables, constructors, and functions mentioned in $I$. We write $I \vdash \diamond$ to mean that the interface $I$ mentions no value, constructor, or function twice, that is, there is no split $I = I', I''$ with $dom(I') \cap dom(I'') \neq \varnothing$. We write $I \vdash \mu$ to mean that $I \vdash \diamond$ and moreover $\mu$ is a member of $I$, that is, $I = I', \mu$ for some $I$'.

The formation judgment $I \vdash S : I'$ means $S$ refers only to external values, constructors, and functions listed in $I$, and provides declarations for the values, constructors, and functions listed in $I'$. The formation judgment $I \vdash e$ means that all occurrences of variables in $e$ are bound and all occurrences of constructors and functions in $e$ have the correct arity. We define these judgments inductively via the rules in the following table. In the rule for **match**, we write $fv(M_i)$:**val** as a shorthand for $x_1$:**val**$, \ldots, x_n$:**val** where $\{x_1, \ldots, x_n\} = fv(M_i)$.

**Formation Rules for F:**

$$\frac{I \vdash x{:}\mathbf{val}}{I \vdash x} \qquad \frac{I \vdash f{:}\mathbf{ctor}\ n \quad I \vdash M_i \quad \forall i \in 1..n}{I \vdash f(M_1,\ldots,M_n)} \qquad \frac{I \vdash \ell{:}\mathbf{fun}\ n \quad I \vdash M_i \quad \forall i \in 1..n}{I \vdash \ell\ M_1\ \ldots\ M_n}$$

$$\frac{I \vdash e}{I \vdash \mathrm{fork}(\mathbf{fun}() \rightarrow e)} \qquad \frac{I \vdash e_1 \quad I, x{:}\mathbf{val} \vdash e_2}{I \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2} \qquad \frac{I \vdash M \quad I, fv(M_i){:}\mathbf{val} \vdash M_i \quad fn(M_i) = \varnothing \quad I, fv(M_i){:}\mathbf{val} \vdash e_i \quad \forall i \in 1..n}{I \vdash \mathbf{match}\ M\ \mathbf{with}\ (\mid M_i \rightarrow e_i)^{i \in 1..n}}$$

$$\frac{I \vdash \diamond}{I \vdash \varnothing : \varnothing} \qquad \frac{I_s = (f_i{:}\mathbf{ctor}\ n_i)^{i \in 1..n} \quad I, I_s \vdash S : I'}{I \vdash \mathbf{type}\ s\ =\ (\mid f_i\ \mathbf{of}\ s_{i1} * \cdots * s_{in_i})^{i \in 1..n}\ S : I_s, I'}$$

$$\frac{I \vdash e \quad I, x{:}\mathbf{val} \vdash S : I'}{I \vdash \mathbf{let}\ x\ = e\ S : x{:}\mathbf{val}, I'} \qquad \frac{I, \ell{:}\mathbf{fun}\ n, x_1{:}\mathbf{val}, \ldots, x_n{:}\mathbf{val} \vdash e \quad I, \ell{:}\mathbf{fun}\ n \vdash S : I'}{I \vdash \mathbf{let}\ \ell\ x_1 \ldots x_n = e\ S : \ell{:}\mathbf{fun}\ n, I'}$$

These formation rules are an abstraction of the typing rules of F# for the fragment we consider. They are enforced by the F# compiler during typechecking.

Recall the system $S_{10} = d_{\mathrm{Ev}}\ d_{\mathrm{Cipher}}\ d_{\mathrm{enc}}\ d_{\mathrm{dec}}\ d_{\mathrm{net}}\ d_{\mathrm{key}}\ d_{\mathrm{init}}\ d_{\mathrm{resp}}\ d_{\mathrm{u1}}\ d_{\mathrm{u2}}$ and the interface Prim given earlier. We can derive that Prim $\vdash S_{10} : I_{10}$, where $I_{10}$ is the interface:

Send: **ctor** 1, Accept: **ctor** 1, Enc: **ctor** 2, enc: **fun** 2, dec: **fun** 2,
net: **val**, key: **val**, init: **fun** 1, resp: **fun** 1, u1: **val**, u2: **val**

*Event-Based Security Properties of F* We express authentication and other properties in terms of event-based queries. The general form of a query is $\mathbf{ev}{:}E \Rightarrow \mathbf{ev}{:}B_1 \vee \cdots \vee \mathbf{ev}{:}B_n$, which means that every reachable configuration containing an event matching the pattern $E$ also contains an event matching one of the $B_i$ patterns.

**Queries and Safety:**

A *query* $q$ is written $\mathbf{ev}{:}E \Rightarrow \mathbf{ev}{:}B_1 \vee \cdots \vee \mathbf{ev}{:}B_n$
for values $E$, $B_1, \ldots, B_n$ containing no free names.
Let $\sigma$ stand for a substitution $\{M_1/x_1, \ldots, M_n/x_n\}$.
Let $C \models \mathbf{query}\ \mathbf{ev}{:}E \Rightarrow \mathbf{ev}{:}B_1 \vee \cdots \vee \mathbf{ev}{:}B_n$ if and only if
whenever $C \equiv \mathbf{event}\ E\sigma \mid C'$, we have $C' \equiv \mathbf{event}\ B_i\sigma \mid C''$ for some $i \in 1..n$.
Let $C \rightarrow^*_{\equiv} C'$ if and only if either $C \equiv C'$ or $C \rightarrow^* C'$.
Let $S$ be *safe for q* if and only if $C \models q$ whenever $S \rightarrow^*_{\equiv} C$.

For example, a system is *safe* for query **ev**:Accept(x) ⇒ **ev**:Send(x) from Section 2 if every reachable configuration containing **event** Accept($M$) also contains **event** Send($M$). Our example system $S_{10}$ satisfies this property. For example, let $C_{10}$ be any one of the configurations shown earlier such that $S_{10} \rightarrow^* C_{10}$. We can easily see that $C_{10} \models$ **ev**:Accept(x) ⇒ **ev**:Send(x), since an Accept event only occurs in the final configuration, which includes a matching Send event.

We define a robust safety property, that is, safety in the presence of an opponent. To avoid vacuous failures, we forbid the opponent from logging events. If $I$ is an interface, an $I$-opponent is a system $O$ that depends only on $I$ and Prim, but not log.

**Formal Threat Model: Opponents and Robust Safety**

Let $S :: I_{pub}$ iff $\text{Prim} \vdash S : I_{pub}, I_{priv}$ for some $I_{priv}$.
Let $O$ be an *$I$-opponent* iff $\text{Prim}\backslash\text{log}, I \vdash O : I'$ for some $I'$.
Let $S$ be *robustly safe for $q$ and $I$* iff $S :: I$ and $S\ O$ is safe for $q$ for all $I$-opponents $O$.

Hence, setting a verification problem for a system $S$ essentially amounts to selecting the subset $I_{pub}$ of its interface that is made available to the opponent.

Consider again our small example $S_{10}$, its interface $I_{10}$, and the query $q$ = **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x) given earlier. We already noted that $S_{10}$ is safe for $q$ and that $\text{Prim} \vdash S_{10} : I_{10}$, but $S_{10}$ is not robustly safe for $q$ and $I_{10}$. The interface $I_{10}$ exposes too much to the opponent, and hence does not reflect our intended threat model. For example, the secret key is included in $I_{10}$, allowing the following opponent $O_1$ to intercept the encrypted message, and replace it with another.

$O_1 \triangleq$ **let** u1 = recv net **let** u2 = send net (enc(`"bogus"`,key))

Moreover, the constructor Enc exposed in $I_{10}$ allows the following opponent $O_2$ to use pattern matching to discover the secret key, and hence to send a bogus message.

$O_2 \triangleq$ **let** u = **match** recv net **with** Enc(m,k) $\rightarrow$ send net (enc(`"bogus"`,k))

The concrete counterpart to this symbolic attack is the ability to extract the encryption key from any ciphertext, a major failure of a cryptosystem. Since this possibility is not normally included in the threat model for protocols, we would not normally export encryption constructors, such as Enc, to the symbolic opponent.

For either $O_1$ or $O_2$ we can calculate the following computation, which ends in a configuration that does not satisfy the query $q$.

$S_{10}\ O_i \rightarrow^+ d_{\text{enc}} \mid d_{\text{dec}} \mid d^{\text{n k}}_{\text{init}} \mid d^{\text{n k}}_{\text{resp}} \mid$ **event** Send(`"msg1"`) $\mid$ **event** Accept(`"bogus"`)

On the other hand, $S_{10}$ is robustly safe for $q$ and the following interface that reflects our intended threat model. The interface does not expose the secret key to the attacker, and by not exporting the constructor Enc prevents the attacker from extracting keys from ciphertexts. It does allow the attacker to initiate protocol roles, to send and receive network traffic, and to encrypt and decrypt messages.

enc: **fun** 2, dec: **fun** 2, net: **val**, init: **fun** 1, resp: **fun** 1

For the example protocol in Section 2, let $S$ be the system that consists of application code and symbolic libraries. We have that $S :: I_{pub}$, where $I_{pub}$ is the example interface given earlier in this section. Our verification problem is to show that $S$ is robustly safe for **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x) and $I_{pub}$.

## 4. Mapping F# to a Verifiable Model

We target the script language of ProVerif for verification purposes. ProVerif can establish correspondence and secrecy properties for protocols expressed in a variant of the pi cal-

culus, whose syntax and semantics are detailed in our technical report. In this calculus, active attackers are represented as arbitrary processes that run in parallel, communicate with the protocol on free channels, and perform symbolic computations. Given a script that defines the protocol, the capabilities of the attacker, and some target query, ProVerif generates logical clauses then uses a resolution-based semi-algorithm. When ProVerif completes successfully, the script is *robustly safe* for the target query, that is, the query holds against all (pi calculus) attackers; otherwise, ProVerif attempts to reconstruct an attack trace. ProVerif may also diverge, or fail, as can be expected since query verification in the pi calculus is not decidable. (ProVerif is known to terminate for the special class of *tagged* protocols [16]. However, the protocols in our main application area of web services rarely fall in this class.) ProVerif is a good match for our purposes, as it offers both general soundness theorems and an effective implementation. Pragmatically, we also rely on previous positive experience in generating large verification scripts for ProVerif. In principle, however, we may benefit from any other verification tool.

To obtain a ProVerif script, we translate F programs to pi calculus processes and rewrite rules. To help ProVerif succeed, we use a flexible combination of several translations. To validate our usage of ProVerif, we also formally relate arbitrary attackers in the pi calculus to those expressible in F.

At its core, our translation maps functions to processes using the classic call-by-value encoding from lambda calculus to pi calculus [31]. For instance, we may translate the mac function declaration of Section 2

```
let mac nonce pwd text =
  Crypto.hmacsha1 nonce (concat (utf8 pwd) (utf8 text))
```

into the process

```
!in(mac, (nonce,pwd,text,k));
 out(k,Hmacsha1(nonce,Concat(Utf8(pwd),Utf8(text))))
```

This process is a replicated input on channel mac; each message on mac carries the functional arguments (nonce,pwd,text) as well as a continuation channel k. When the function completes, it sends back a message that carries its result on channel k. Similarly, we translate the server function declaration of Section 2 into:

```
!in(server, (arg,kR));
 new kX; out(accept, (address,kX)); in(kX,xml);
 new kM; out(unmarshall, (xml,kM)); in(kM,(m,en,text));
 new kV; out(verify, ((m,en,text),sk,pwd,kV)); in(kV,());
 event Ev(Accept(text));
 out(kR, ())
```

This process first calls function accept as follows: it generates a fresh continuation channel kX; it sends a message that carries the argument address and kX on channel accept; and it receives the function result xml on channel kX. The process then similarly calls the functions unmarshall and verify. If both calls succeed, the process finally logs the event Accept(text) and returns an (empty) result on kR.

Our pi calculus includes the same term algebra—values built from variables, names, and constructors—as F, so values are unchanged by the translation. Moreover, our pi calculus includes term destructors defined by rewrite rules on the term algebra, and whenever possible after inlining, our implementation maps simple functions to destructors. For instance, we actually translate the mac function declaration into the native ProVerif reduction:

```
reduc mac(nonce,pwd,text) =
  HmacSha1(nonce,Concat(Utf8(pwd),Utf8(text)))
```

Both formulations of mac are equivalent, but the latter is more efficient. On the other hand, complex functions with side-effects, recursion, or non-determinism are translated as processes. Our tool also supports a third potential translation for mac, into a ProVerif predicate declaration; predicates are more efficient than processes and more expressive than reductions. Our translation first performs aggressive inlining of F functions, constant propagation, and similar optimizations. It then globally picks the best applicable formulation for each reachable function, while eliminating dead code.

Finally, the translation gives to the pi calculus context the capabilities available to attackers in F. For example, the channel httpchan representing network communication is exported to the context in an initialization message. More interestingly, every public function coded as a process is made available on an exported channel.

For instance, the server function is available to the attacker; accordingly, we generate the process:

```
!in(serverPUB, (arg,kR)); out(server, (arg,kR))
```

This enables the attacker to trigger instances of the server using the public channel serverPUB. Conversely, the private channel server is used only by the translation, so that the attacker cannot intercept local function calls.

Formally, we define translations for expressions $e$, declarations $d$, and systems $S$. The translation $\mathscr{E}[\![e]\!](x,P)$ is a process that binds variable $x$ to the value of $e$ and then runs process $P$. The translations $\mathscr{S}[\![d]\!](P)$ and $\mathscr{S}[\![S]\!](P)$ are processes that elaborate $d$ and $S$, and then run process $P$. At the top level, the translation $[\![S :: I_{pub}]\!]$ is a ProVerif script that includes constructor definitions for the datatypes in $S$ and defines a process that elaborates $S$ and then exports $I_{pub}$. Details of these translations are in the technical report.

Our main correctness result is the following.

**Theorem 1 (Reflection of Robust Safety)** *If $S :: I_{pub}$ and $[\![S :: I_{pub}]\!]$ is robustly safe for $q$, then $S$ is robustly safe for $q$ and $I_{pub}$.*

In the statement of the theorem, $S$ is the series of modules that define our system; $I_{pub}$ is a selection of the values, constructors, and functions declared in $S$ that are made available to the attacker; $q$ is our target security query; and $[\![S :: I_{pub}]\!]$ is the ProVerif script obtained from $S$ and $I_{pub}$.

The proof of Theorem 1 appears in our technical report; it relies on an operational correspondence between reductions on F configurations and reductions in the pi calculus.

We implement our translation as a command line tool fs2pv that intercepts code after the F# compiler front-end. The tool takes as input a series of module implementations defining $S$ and module interfaces bounding the attacker's capabilities, much like $I_{pub}$. The tool relies on the typing discipline of F# (which is stronger than the scope discipline of F) to enforce that $S :: I_{pub}$. It then generates the script $[\![S :: I_{pub}]\!]$ and runs ProVerif. If ProVerif completes successfully, it follows that $[\![S :: I_{pub}]\!]$ is robustly safe for $q$. Hence, by Theorem 1, we conclude that $S$ is robustly safe for $q$ and $I_{pub}$.

As a simple example, recall the system $S$ and its interface $I_{pub}$, as stated at the end of Section 3. Our tool runs successfully on this input, proving that $S$ is robustly safe for the query **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x) and $I_{pub}$.

## 5. Verification of Interoperable Code

To validate our approach experimentally, we implemented a series of cryptographic protocols and verified their security against demanding threat models.

Tables 1 and 2 summarize our results for these protocols. For each protocol, Table 1 gives the program size for the implementation (in lines of F# code, excluding interfaces and code for shared libraries), the number of messages exchanged, and the size of each message, measured both in bytes for concrete runs and in number of constructors for symbolic runs. Table 2 concerns verification; it gives the number of queries and the kinds of security properties they express. A secrecy query requires that a password (pwd) or key (key) be protected; a weak-secrecy query further requires that a weak secret (weak pwd) be protected from a guessing attack. An authentication query requires that a message content (msg), its sender (sender), or the whole exchange (session) be authentic. Some queries can be verified even in the presence of attackers that control some corrupted principals, thereby getting access to their keys and passwords. Not all queries hold for all protocols; in fact some queries are designed to test the boundaries of the attacker model and are meant to fail during verification. Finally, the table gives the size of the logical model generated by ProVerif (the number of logical clauses) and its total running time to verify all queries for the protocol.

For example, consider the simple authentication protocol of Section 2, named *Password-based MAC* in the tables; its implementation has 38 lines of specific code; ProVerif takes less than one second to verify the message authentication query and to verify that the protocol protects the password from guessing attacks. A variant of our implementation for this protocol (second row of Tables 1 and 2) produces the same message, but is more modular and relies on more realistic libraries; it supports distributed runs and enables the verification of queries against active attackers that may selectively corrupt some principals and get access to their keys and passwords.

As a benchmark, we wrote a program for the four message Otway-Rees key establishment protocol [36], with two additional messages after key establishment to probe the secrecy of message payloads encrypted with this key. To complete a concrete, distributed implementation, we had to code detailed message formats, left ambiguous in the description of the protocol. In the process, we inadvertently enabled a typing attack, immediately found by verification. We experimented with a series of 16 authentication and secrecy queries; their verification takes a few minutes.

| Protocol | Implementation | | | |
|---|---|---|---|---|
| | LOCs | messages | bytes | symbols |
| *Password-based MAC* | 38 | 1 | 208 | 16 |
| *Password-based MAC variant* | 75 | 1 | 238 | 21 |
| *Otway-Rees* | 148 | 4 | 74; 140; 134; 68 | 24; 40; 20; 11 |
| *WS password-based signing* | 85 | 1 | 3835 | 394 |
| *WS X.509 signing* | 85 | 1 | 4650 | 389 |
| *WS password-based MAC* | 85 | 1 | 6206 | 486 |
| *WS request-response* | 149 | 2 | 6206; 3187 | 486; 542 |

**Table 1.** Summary of example protocols

| Protocol | Security Goals | | | | Verification | |
|---|---|---|---|---|---|---|
| | queries | secrecy | authentication | insiders | clauses | time |
| *Password-based MAC* | 4 | weak pwd | msg | no | 69 | 0.8s |
| *Password-based MAC variant* | 5 | pwd | msg, sender | yes | 213 | 2.2s |
| *Otway-Rees* | 16 | key | msg, sender | yes | 155 | 1m50s |
| *WS password-based signing* | 5 | no | msg, sender | yes | 456 | 5.3 s |
| *WS X.509 signing* | 5 | no | msg, sender | yes | 460 | 2.6 s |
| *WS password-based MAC* | 3 | weak pwd | msg, sender | no | 436 | 10.9s |
| *WS request-response* | 15 | no | session | yes | 503 | 44m45s |

**Table 2.** Verification Results

*A Library for Web Services Security* As a larger, more challenging case study, we implemented and verified several web services security protocols.

Web services are applications that exchange XML messages conforming to the SOAP standard [25]. To secure these exchanges, messages may include a security header, defined in the WS-Security standard [34], that contains signatures, ciphertexts, and a range of security elements, such as tokens that identify particular principals. Hence, each secure web service implements a security protocol by composing mechanisms defined in WS-Security. Previous analyses of such WS-Security protocols established correctness theorems [23,9,7,27,28] and uncovered attacks [9,11]. However, these analyses operated on models of protocols and not on their implementations. In the rest of this section, we present the first verification results for the security of interoperable web services implementations.

First, we develop a library in F that implements the formats and mechanisms of the web services messaging and security specifications. Like WSE [30], our library is a partial implementation of these specifications; we selected features based on the need to interoperate with protocols implemented by WSE. Our library provides several modules:

- Soap implements the SOAP formats for requests, responses, and faults, and their exchange via HTTP.
- Wsaddressing implements the WS-Addressing [17] header formats, for message routing and correlation.

- Xmldsig and Xmlenc implement the standards for XML digital signature [20] and XML encryption [19], which provide flexible formats for selectively signing and encrypting parts of an XML document.
- Wssecurity implements the WS-Security header format and common security tokens, such as username tokens, encrypted keys, and X.509 certificates.

These modules rely on the Crypto module for cryptographic functions and a new Xml module (with dual symbolic and concrete implementations) for raw XML manipulation.

Applications written with this library produce and consume SOAP messages that conform to the web services specifications. Such applications can interoperate with other conformant web services, such as those that use WSE.

The requirement to produce concrete, interoperable, and verifiable code is quite demanding, but it yields very precise executable models for the informal WS-Security specifications, more detailed than any available in the literature. For verifiability, we adopt a programming discipline that reduces the flexibility of message formats wherever possible. In particular, we fix the order of headers in a message and limit the number of headers that can be signed. We avoid higher-order functions (such as List.map) and recursion over lists and XML, and instead inline these functions by hand.

The library consists of 1200 lines of F code. We can quickly write security protocols using this library, such as an authentication protocol that uses a password or an X.509 certificate to generate an XML digital signature (protocols *WS Password-based signing* and *WS X.509 signing* in Tables 1 and 2). Only 85 additional lines of code need to be written to implement these protocols; their verification takes a few seconds.

*A Simple Authentication Protocol over WS-Security* As a case study, we used our web services library to implement an existing password-based authentication protocol (*WS password-based MAC*) taken from the WSE samples. The protocol is quite similar to *Password-based MAC*, except that the message is now a standards-compliant XML document. This message is sent as the body of a SOAP envelope that includes a WS-Security security header that contains a *username token*, representing the client's identity, and an *X.509 token*, representing the server's identity. The username token includes a freshly generated nonce used, along with a shared password, to derive a key for message authentication. This nonce is protected by encrypting the entire username token with the server's public key, using XML encryption. The message is authenticated by an XML digital signature that includes a cryptographic keyed hash of the body using a key derived from the username token.

In earlier work [11], we wrote a non-executable formal model for this protocol and analyzed it with ProVerif. Here, we extract the model directly from a full-fledged implementation. Moreover, we encode a more realistic threat model that enables the attacker to gain access to some passwords and keys. In particular, the Prins module has two additional functions in its interface: leakPassword and leakPrivateKey.

The leakPassword function is defined as follows:

```
let leakPassword (u:str) =
  let pwd = getPassword u in log Leak(u); pwd
```

When the attacker calls leakPassword for a principal u, the function extracts the password for u from the database and returns it to the attacker; but before leaking the

password, the function logs an event Leak(u) recording that the principal u has been compromised.

We implement the client and server roles using our library, with slightly different Send and Accept events from the ones in Section 2. To enable sender authentication, the client logs Send(u,m), where u is the principal that sends the XML message m. Similarly, on receiving the message, the server logs Accept(u,m). The datatype of events and the authentication query becomes

**type** Ev = Send **of** str∗item
 | Accept **of** str∗item
 | Leak **of** str
q = **ev**:Accept(u,m) ⇒ **ev**:Send(u,m) ∨ **ev**:Leak(u)

where item is the datatype of XML elements. The query q asks that the server authenticate the message m and the sending principal u, unless u has been leaked. Let $W$ be the system that consists of the client and server code, the symbolic libraries (Crypto, Net, Prins, and Xml), and the web services library. Let $I_{pub}$ be the interface of Section 3 extended with the item datatype. Using fs2pv and ProVerif, we prove that $W$ is robustly safe for q and $I_{pub}$. The verification of message and sender authentication takes only a few seconds. As with *Password-based MAC*, we also prove that the password is protected even if it is a weak secret.

We experimentally checked that our concrete implementation complies with the web services specifications: we can run our client with a WSE server, and conversely access our server from a WSE client. Many details of our model would have been difficult to determine from the specifications alone, without interoperability testing. The resulting messages exchanged by the concrete execution are around 6 kilobytes in size, while the symbolic execution of the protocol generates messages with 486 symbols. The performance of our concrete implementation is comparable to WSE, which is not surprising here, since the execution time is dominated by XML processing and communication.

We also implemented and verified an extension of the protocol described above, where the server, upon accepting the request message, sends back a response message signed with the private key associated with its X.509 certificate. For this two message protocol, the security goals are authentication of the request and the response, as well as correlation between the messages. Correlation relies on a mechanism called *signature confirmation* (described in a draft revision of WS-Security), where the response echoes and signs the password-based signature value of the request. The protocol is named *WS request-response* in the tables; ProVerif establishes all our authentication and correlation goals, but takes almost 45 minutes for the analysis. Elsewhere [10], we describe the design and architecture of the library used for this and other web services security protocols.

Our protocol implementation can also be used as part of a larger web application, while still benefiting from our results. The client functions can be exported as a library invoked by applications written in any language running on the CLR, such as C# or Visual Basic. Similarly, the server functions can be embedded in the security stack of a web server that checks all incoming messages for conformance to the protocol before handing over the message body to a web application written in any language. In both cases, assuming the application code does not have access to secret passwords or keys, the security results transparently apply.

## 6. Conclusions

We describe an architecture and programming model for security protocols. For production use, protocol code runs against concrete cryptography and low-level networking libraries. For initial development, the same code runs against symbolic cryptography and intra-process communication libraries. For verification, much of the code translates to a low-level pi calculus model for analysis against a Dolev-Yao attacker. The attacker can be understood and customized in source-level terms as an arbitrary program running against an interface exported by the protocol code.

Our prototype implementation is the first, we believe, to extract verifiable models from code implementing standard security protocols, and hence able to interoperate with other implementations. Our prototype has many limitations; still, we conclude that it significantly reduces the gap between symbolic models of cryptographic protocols and their implementations.

*Limits of our model* As usual, formal security guarantees hold only within the boundaries of the model being considered. Automated model extraction, such as ours, enables the formal verification of large, detailed models closely related to implementations. In our experience, such models are more likely to encompass security flaws than those focusing on protocols in isolation. Independently of our work, modelling can be refined in various directions. Certified compilers and runtime environments can give strong guarantees that program executions comply with their formal semantics; in our setting, they may help bridge the gap between the semantics of F and a low-level model of its native-code execution, dealing for instance with memory safety.

Our approach also crucially relies on the soundness of symbolic cryptography with regards to one implementation of concrete cryptography, which is far from obvious. Pragmatically, our modelling of symbolic cryptography is flexible enough to accommodate many known weaknesses of cryptographic algorithms (introducing for instance symbolic cryptographic functions "for the attacker only"). There is a lot of interesting research on reconciling symbolic cryptography with more precise computational models [3,6]. Still, for the time being, these models do not support automated analyses on the scale needed for our protocols.

*Related work* The ideas of modelling protocol roles as functions and modelling an active attacker as an arbitrary functional context appear earlier in Sumii and Pierce's studies of cryptographic protocols within a lambda calculus [39,40]. Unlike our functional language, which has state and concurrency, their calculus cannot directly capture linearity properties (such as replay detection via nonces), as its only imperative feature is name generation. Several systems [37,33,29,38] operate in the reverse direction, and generate runnable code from abstract models of cryptographic protocols in formalisms such as strand spaces, CAPSL, and the spi calculus. These systems need to augment the underlying formalisms to express implementation details that are ignored in proofs, such as message sizes and error handlers. Going further in the direction of growing a formalism into a programming language, Guttman, Herzog, Ramsdell, and Sniffen [26] propose a new programming language CPPL for writing security protocols; CPPL combines features for communication and cryptography with a trust management engine for logically-defined authorization checks. CPPL programs can be verified using strand space techniques, although there is no automatic support for this at present. A limitation of all of

these systems is that they do not implement standard message formats and hence do not interoperate with other implementations. In terms of engineering effort, it seems easier to achieve interoperability by starting from an existing general purpose language such as F# than by developing a new compiler.

Giambiagi and Dam [22] take a different approach to showing the conformance of implementation to model. They neither translate model to code, nor code to model. Instead, they assume both are provided by the programmer, and develop a theory to show that the information flows allowed by the implementation of a cryptographic protocol are none other than those allowed by the abstract model of the protocol. They treat the abstract protocol as a specification for the implementation, and implicitly assume correctness of the abstract protocol.

Askarov and Sabelfeld [5] report a substantial distributed implementation within the Jif security-typed language of a cryptographic protocol for online poker without a trusted third party. Their goal is to prevent some insecure information flows by typing. They do not derive a formal model of the protocol from their code.

There are only a few works on compiling implementation files for cryptographic protocols to formal models. Bhargavan, Fournet, and Gordon [8] translate the policy files for web services to the TulaFale modelling language [11], for verification by compilation to ProVerif. This translation can detect protocol errors in policy settings, but applies to configuration files rather than executable source code. Other symbolic modelling [23,9, 7,27,28] of web services security protocols has uncovered a range of potential attacks, but has no formal connection to source code. Goubault-Larrecq and Parrennes [24] are the first to derive a Dolev-Yao model from implementation code written in C. Their tool Csur performs an interprocedural points-to analysis on C code to yield Horn clauses suitable for input to a resolution prover. They demonstrate Csur on code implementing the initiator role of the Needham-Schroeder public-key protocol.

There is also recent research on verifying implementations of cryptographic algorithms, as opposed to protocols. For instance, Cryptol [21] is a language-based approach to verifying implementations of algorithms such as AES.

*Acknowledgements* James Margetson and Don Syme helped us enormously with using and adapting the F# compiler. Tony Hoare and David Langworthy suggested improvements to the presentation.

## References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[4] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 140–154, 2005.

[5] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *10th European Symposium on Research in Computer Security (ESORICS'05)*, volume 3679 of *LNCS*, pages 197–221. Springer, 2005.

[6] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 220–230. ACM Press, 2003.

[7] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In *2004 ACM Workshop on Secure Web Services (SWS)*, pages 11–22, Oct. 2004.

[8] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, Oct. 2004.

[9] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. *Theoretical Comput. Sci.*, 340(1):102–153, June 2005.

[10] K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of ws-security protocols. In *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *LNCS*, pages 88–106. Springer, 2006.

[11] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 197–222. Springer, 2004.

[12] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.

[13] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. Technical Report MSR–TR–2006–46, Microsoft Research, 2006.

[14] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

[15] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 331–340, 2005.

[16] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, 2005.

[17] D. Box, F. Curbera, et al. *Web Services Addressing (WS-Addressing)*, Aug. 2004. W3C Member Submission.

[18] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[19] D. Eastlake, J. Reagle, et al. *XML Encryption Syntax and Processing*, 2002. W3C Recommendation.

[20] D. Eastlake, J. Reagle, D. Solo, et al. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation.

[21] Galois Connections. *Cryptol Reference Manual*, 2005.

[22] P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Science of Computer Programming*, 50:73–99, 2004.

[23] A. D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *2002 ACM workshop on XML Security*, pages 18–29, 2002.

[24] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.

[25] M. Gudgin et al. *SOAP Version 1.2*, 2003. W3C Recommendation.

[26] J. D. Guttman, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Programming cryptographic protocols. In *Trusted Global Computing (TGC'05)*, volume 3705 of *LNCS*, pages 116–145. Springer, 2005.

[27] E. Kleiner and A. W. Roscoe. Web services security: A preliminary study using Casper and FDR. In *Automated Reasoning for Security Protocol Analysis (ARSPA 04)*, 2004.

[28] E. Kleiner and A. W. Roscoe. On the relationship between web services security and traditional protocols. In *Mathematical Foundations of Programming Semantics (MFPS XXI)*, 2005.

[29] S. Lukell, C. Veldman, and A. C. M. Hutchison. Automated attack analysis and code generation in a multi-dimensional security protocol engineering framework. In *Southern African Telecommunication Networks and Applications Conference (SATNAC)*, 2003.

[30] Microsoft Corporation. *Web Services Enhancements (WSE) 2.0*, 2004. At `http://msdn.microsoft.com/webservices/building/wse/default.aspx`.

[31] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[32] R. Milner. *Communicating and Mobile Systems: the $\pi$-Calculus*. CUP, 1999.

[33] F. Muller and J. Millen. Cryptographic protocol generation from CAPSL. Technical Report SRI–CSL–01–07, SRI, 2001.

[34] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, Mar. 2004. OASIS Standard 200401.

[35] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers.

*Commun. ACM*, 21(12):993–999, 1978.

[36] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operation Systems Review*, 21(1):8–10, 1987.

[37] A. Perrig, D. Song, and D. Phan. AGVI – automatic generation, verification, and implementation of security protocols. In *13th Conference on Computer Aided Verification (CAV)*, LNCS, pages 241–245. Springer, 2001.

[38] D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004)*, volume 1, pages 400–405, 2004.

[39] E. Sumii and B. C. Pierce. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 256–269, 2001.

[40] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 161–172, 2004.

[41] D. Syme. *F#*, 2005. Project website at `http://research.microsoft.com/fsharp/`.

[42] T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*

# Compensable transactions

Tony HOARE
*Microsoft Research, Cambridge, England*

**Abstract.** The concept of a compensable transaction has been embodied in modern business workflow languages like BPEL. This article uses the concept of a box-structured Petri net to formalise the definition of a compensable transaction. The standard definitions of structured program connectives are extended to construct longer-running transactions out of shorter fine-grain ones. Floyd-type assertions on the arcs of the net specify the intended properties of the transaction and of its component programs. The correctness of the whole transaction can therefore be proved by simple local reasoning.

## 1. Introduction

A compensable transaction can be formed from a pair of programs: one that performs an action and another that performs a compensation for that action if and when required. The forward action is a conventional atomic transaction: it may fail before completion, but before failure it guarantees to restore (an acceptable approximation of) the initial state of the machine, and of the relevant parts of the real world. A compensable transaction has an additional property: after successful completion of the forward action, a failure of the next following transaction may trigger a call of the compensation, which will undo the effects of the forward action, as far as possible. Thus the longer transaction (this one together with the next one) is atomic, in the sense that it never stops half way through, and that its failure is adequately equivalent to doing nothing. In the (hopefully rare) case that a transaction can neither succeed nor restore its initial conditions, an explicit exception must be thrown.

The availability of a suitable compensation gives freedom to the forward action to exercise an effect on the real world, in the expectation that the compensation can effectively undo it later, if necessary. For example, a compensation may issue apologies, cancel reservations, make penalty payments, etc. Thus compensable transactions do not have to be independent (in the sense of ACID); and their durability is obviously conditional on the non-occurrence of the compensation, which undoes them. Because all our transactions are compensable, in this article we will often omit the qualification.

We will define a number of ways of composing transactions into larger structures, which are also compensable transactions. Transaction declarations can even be nested. This enables the concept of a transaction to be re-used at many levels of granularity, ranging perhaps from a few microseconds to several months – twelve orders of magnitude. Of course, transactions will only be useful if failure is rare, and the longer transactions must have much rarer failures.

The main composition method for a long-running transaction is sequential composition of an ordered sequence of shorter transactions. Any action of the sequence may fail,

and this triggers the compensations of the previously completed transactions, executed in the reverse order of finishing. A sequential transaction succeeds only if and when all its component transactions have succeeded.

In the second mode of composition, the transactions in a sequence are treated as alternatives: they are tried one after another until the first one succeeds. Failure of any action of the sequence triggers the forward action of the next transaction in the sequence. The sequence fails only if and when all its component transactions have failed.

In some cases (hopefully even rarer than failure), a transaction reaches a state in which it can neither succeed nor fail back to an acceptable approximation of its original starting state. The only recourse is to throw an exception. A catch clause is provided to field the exception, and attempt to rectify the situation.

The last composition method defined in this article introduces concurrent execution both of the forward actions and of the backward actions. Completion depends on completion of all the concurrent components. They can all succeed, or they can all fail; any other combination leads to a throw.

## 2. The Petri box model of execution

A compensable transaction is a program fragment with several entry points and several exits. It is therefore conveniently modelled as a conventional program flowchart, or more generally as a Petri net. A flowchart for an ordinary sequential program is a directed graph: its nodes contain programmed actions (assignments, tests, input, output, . . . as in your favourite language), and its arrows allow passage of a single control token through the network from the node at its tail to the node at its head. We imagine that the token carries with it a value consisting of the entire state of the computer, together with the state of that part of the world with which the computer interacts. The value of the token is updated by execution of the program held at each node that it passes through. For a sequential program, there is always exactly one token in the whole net, so there is never any possibility that two tokens may arrive at an action before it is complete.

In section 6, we introduce concurrency by means of a Petri net transition, which splits the token into separate tokens, one for each component thread. It may be regarded as carrying that part of the machine resources which is owned by the thread, and communication channels with those parts of the real world for which it is responsible. The split token is merged again by another transition when all the threads are complete. The restriction to a single token therefore applies within each thread.

A structured flowchart is one in which some of its parts are enclosed in boxes. The fragment of a flowchart inside a box is called a block. The perimeter of a box represents an abstraction of the block that it contains. Arrows crossing the perimeter are either entries or exits from the box. We require the boxes to be either disjoint or properly nested within each other. That is why we call it a structured flowchart, though we relax the common restriction that each box has only one entry and one exit arrow. The boxes are used only as a conceptual aid in planning and programming a transaction, and in defining a calculus for proving their correctness. In the actual execution of the transaction, they are completely ignored.

We will give conventional names to the entry points and exit points of the arrows crossing the perimeter of the box. The names will be used to specify how blocks are com-

posed into larger blocks by connecting the exits of one box to the entries of another, and enclosing the result in yet another box. This clearly preserves the disjointness constraint for a box-structured net.

One of the arrows entering the box will be designated as the start arrow. That is where the token first enters the box. The execution of the block is modelled by the movement of the token along the internal arrows between the nodes of the graph that are inside the box. The token then can leave the box by one of its exit points, generally chosen by the program inside the box. The token can then re-enter the box again through one of the other entry points that it is ready to accept it. The pattern of entering and leaving the block may be repeated many times.

In our formal definition of a compensable transaction, we will include a behavioural constraint, specifying more or less precisely the order in which entry and exit points can be activated. The behavioural constraint will often be expressed as a regular expression, whose language defines all permissible sequences of entry and exit events which may be observed and sequentially recorded.

We will introduce non-determinism into our flowchart by means of the Petri net place. A place is drawn as a small circle (Figure 1) with no associated action. It may have many incoming arrows and many outgoing arrows. The place is entered by a token arriving along any one of its entries. The next action (known as a firing) of the place is to emit the token just once, along any one of its exit arrows. The token arriving at the exit of the place may have originated at any one of its entries. The strict alternation of entries and exits of a place may be formally described by the regular expression

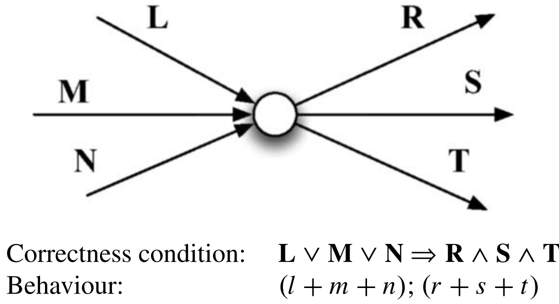
$$(l + m + n); (r + s + t)$$

where $l, m, n$ name the entries of the place, and $r, s, t$ name the exits.

Technical note: in general, a Petri net place is capable of storing a token. In our restricted calculus this capability is exploited only once (in section 6). In fact, we may regard a token as passing instantaneously (as a single event) through any sequence of consecutive states. Of course, a regular expression cannot model this simultaneity.

If the place has only a single exit arrow, it acts as a normal fan-in, and has the same function as in a conventional flowchart. If there are many exits, the place acts as a fan-out. The choice of exit arrow on each occasion of entry is usually determined by the current readiness of the block at the head of the exit arrow to accept entry of the token. But if more than one block is ready, the choice is non-deterministic, in the usual sense of don't-care or demonic non-determinism. It is the programmer's responsibility to ensure that all choices are correct; and the implementation may choose any alternative according to any criterion whatsoever, because it is known that correctness will not be affected. For example, efficiency and responsiveness are among the more desirable of the permissible criteria.

We follow Floyd's suggestion that the arrows in a flowchart should be annotated with assertions. Assertions are descriptions of the state of the world (including the state of the machine), and the programmer intends that they should be true of the world value carried by the control token, whenever it passes along the annotated arrow. An assertion on an arrow which enters a box serves as a precondition for the block, and it is the responsibility of the surrounding flowchart to make it true before transmitting the token to that entry. An assertion on an exit arrow from the box serves as a postcondition, and

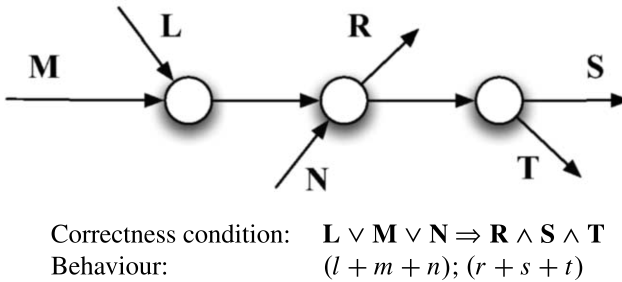



Correctness condition: $\mathbf{L} \vee \mathbf{M} \vee \mathbf{N} \Rightarrow \mathbf{R} \wedge \mathbf{S} \wedge \mathbf{T}$
Behaviour: $(l + m + n); (r + s + t)$

**Figure 1.** A Petri net place

it is the responsibility of the block itself to make it true before transmitting the token through that exit.

The correctness of the composed flowchart may be determined locally in the usual way, by considering the assertions on each arrow and on each place. For an arrow which connects a single exit point to a single entry (usually on another box), the exit assertion of the box at the tail of the arrow must logically imply the entry assertion of the box at its head. For a place, the rule is a natural extension of this. A place is correct if the assertion on each one of its entry arrows logically implies every one of the assertions at the heads of its exit arrows. In other words, the verification condition for a place is that the disjunction of all the tail assertions implies the conjunction of all the head assertions (see Figure 1, where the upper case letters stand for the arrows annotated by the corresponding lower case letter). Thus overall correctness of the entire flowchart can be proved in a modular fashion, just one arrow or place or action at a time.




Correctness condition: $\mathbf{L} \vee \mathbf{M} \vee \mathbf{N} \Rightarrow \mathbf{R} \wedge \mathbf{S} \wedge \mathbf{T}$
Behaviour: $(l + m + n); (r + s + t)$

**Figure 2.** The same place as Figure 1

The intention of drawing a box of a structured flowchart is that the details of the flowchart inside the box should be irrelevant to the rest of the flowchart that lies outside the box. From the outside, you only need to know three items: (1) the names of the entry and exit points; these are used to specify how the box is connected into its environment (2) the assertions on the arrows that enter and leave the box; and (3) the constraints that govern the order of entry and exit events, by which the token enters and leaves the box along the arrows that cross the perimeter. If two boxes have the same assertions and the same set of behaviours, we define them to be semantically equivalent.

This rule of equivalence may be applied to just a single place. As a result, any complete collection of linked Petri net places, in which all the entries are connected to all

the exits, can be replaced by a single place, – one that has all the same entries and the exits, but the internal arrows areeliminated. Figure 2 therefore has the same semantics as Figure 1.

## 3. Definition of a transaction

A compensable transaction is a special kind of box in a Petri net. It is defined as a box whose names, behaviour and assertions satisfy a given set of constraints. The first constraint is a naming constraint. A transaction box has two entry points named *start* and *failback*, and three exit points named *finish*, *fail* and *throw*. The intended function of each of these points is indicated by its name, and will be more precisely described by the other constraints. When a transaction is represented as a box, we introduce the convention that these entries and exits should be distributed around the perimeter as shown in Figure 3. As a result, our diagrams will usually omit the names, since the identity of each arrow is indicated by its relative position on the perimeter of the box.

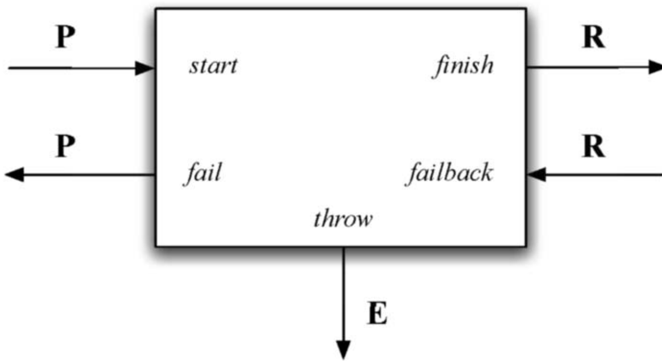



**Figure 3.** Entry and exit names

A more significant part of the formal definition of a transaction is a behavioural constraint, constraining the order in which the token is allowed to enter and exit the block at each entry and exit point. The constraint is conveniently defined by a regular expression:

*start ; (finish ; failback)* ; (fail + throw + finish)*

This expression stipulates that the first activation of the transaction is triggered by entry of the token at the *start* point. The last de-activation of the transaction is when the token leaves at any one of the three exit points. In between these two events, the transaction may engage in any number of intermediate exits and entries. On each iteration, it finishes successfully, but is later required to compensate by a *failback* entry, triggered by failure of the following sequentially composed transaction. The number of occurrences of *finish* followed by *failback* is not limited, and may even be zero. Typical complete behaviours of a transaction are:

*start, finish*
*start, finish, failback, fail*
*start, finish, failback, finish*
*start, finish, failback, finish, failback, throw*

The final constraint in the definition of a transaction governs the assertions on its entry and exit points. This constraint expresses the primary and most essential property of a transaction: that if it fails, it has already returned the world to a state that is sufficiently close to the original initial state.

Sufficient closeness might be defined in many ways, but we give the weakest reasonable definition. Our simple requirement is that on failure the world has been returned to a state which again satisfies the initial precondition of the transaction. More precisely, the assertion on the *fail* exit, must be the same original precondition that labels the start entry point. Similarly, on *failback* the transaction may assume that the postcondition that it previously established on finishing is again valid. These standard assertional constraints are indicated by the annotations in Figure 3. There is no constraint on the assertion **E** labelling the *throw* exit.

Many of the constructions of our calculus of transactions can be applied to transactions which satisfy weaker or stronger assertional constraints than the standard described above. For example, a transaction may be *exactly* compensable if on failure it returns to exactly the original state (where obviously the state of the world must be defined to exclude such observations as the real time clock). A weaker constraint is that the postcondition of failure is merely implied by the precondition. Finally, there is the possibility that the transaction has no assertional constraint at all. We will not further consider these variations.

In drawing diagrams with many boxes, the assertions on the arrows will often be omitted. It is assumed that in concrete examples they will be restored in any way that satisfies the intended assertional constraint, and also satisfies the local correctness criterion for assertions, which apply to all arrows and all places.

That concludes our semantic definition of the concept of a transaction. The flowchart gives an operational semantics, describing how the transactions are executed. The assertions give an axiomatic semantics, describing how the transactions are specified and proved correct. The interpretation of a flowchart makes it fairly obvious that the operational and the axiomatic definitions are in close accord.

## 4. A Calculus of Transactions

In this section we will define a small calculus for design and implementation of transactions. They are built up by applying the operators that we define to smaller by building them from smaller component transactions. The ultimate components are ordinary fragments of sequential program. Our semantic definitions will mainly use the pictorial representation shown in Figure 3. But for the sake of completeness, here is a more conventional syntax.

*<transaction>* ::= *<composed transaction>* | *<primitive transaction>*
*<primitive transaction>* ::= ***succeed*** | ***fail*** | ***throw*** | *<transaction declaration>*
*<transaction declaration>* ::= [*<forward action>* ***comp*** *compensation>*]
*<forward action>* ::= *<ordinary program>*
*<compensation>* ::= *<ordinary program>*
*<composed transaction>* ::= *<sequential composition>* |
    *<alternative composition>* | *<exception block>* | *<non-deterministic choice>*
*<sequential composition>*::= *<transaction>* **;** *<transaction>*
*<alternative composition>* ::= *<transaction>* ***else*** *<transaction>*
*<exception block>* ::= *<transaction>* ***catch*** *<transaction>*
*<non-deterministic choice>* ::= *<transaction>* ***or*** *<transaction>*

The shortest primitive transactions are those that do nothing. There are three ways of doing nothing: by succeeding, by failing, or by throwing. Definitions for these transactions are given diagrammatically in Figure 4, where the small unconnected arrows will never be activated. The leftmost example does nothing but succeed, and this can obviously be compensated by doing nothing again. The other two examples do nothing but fail or throw. These will never be called upon to compensate.

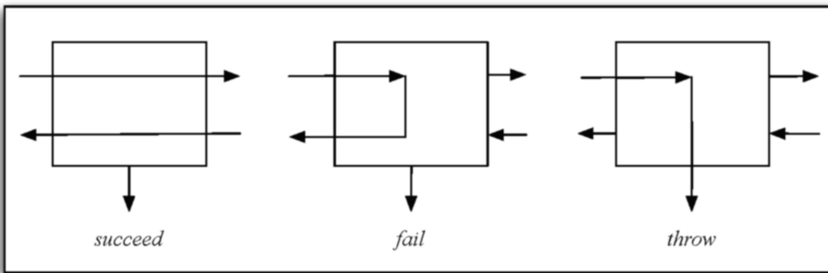



**Figure 4.** Primitive Transactions

Longer running transactions are constructed by composing smaller ones in sequence, or as alternatives, or as a non-deterministic choice, or by try/catch clauses. In all cases, the result of the composition of compensable transactions will also be a compensable transaction. The semantics of each construction will be explained diagrammatically as a box that encloses the boxes representing the components. Some of the exit arrows of each component box will be connected to some of the entry arrows of the other component box, and thereby become internal arrows that can be ignored from outside. Entries and exits on the surrounding box are connected to remaining exits and entries of the component boxes, often ones that share the same name. Where necessary, places may be introduced to deal with fan-in and fan-out.

The basic primitive fine-grained transaction is declared by specifying two sections of normal sequential code (Figure 5). The first of them **T** performs the required action as control passes from the *start* on the left to the *finish* on the right. The second section of code **U** specifies how the action should be compensated as control passes back from the *failback* on the right to the *fail* on the left. Either the action or the compensation can throw, on detecting that neither progress nor compensation is possible. The fan-in

of the *throw* arrow indicates that it is not known from the outside which of the two components has actually performed the throw. This fan-in of throws is common to most of the operators defined below, and will sometimes be omitted from the diagrams

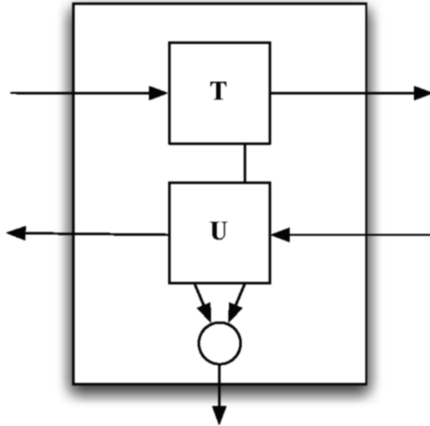



**Figure 5.** Transaction Declaration: **[ T *comp* U ]**

The first definition of the constructions for non-primitive transactions will be sequential composition, which is shown in Figure 6. The outer block denoting the whole composition starts with the *start* of the first component block **T**. The *finish* of this block triggers the *start* of the second component block **U**. The *finish* of the second block finishes the whole sequential composition. A similar story can be told of the backward-going failure path, which performs the two compensations in the reverse order to the forward operations. This is what makes the composed transaction compensable in the same way as its components are. Furthermore, the sequential composition will satisfy the behavioural constraint for transactions, simply because its components do so.

There should be assertions on each of the arrows. However, the permitted patterns for these assertions are completely determined by the correctness principle for flowcharts, so there is no need to mention them explicitly.

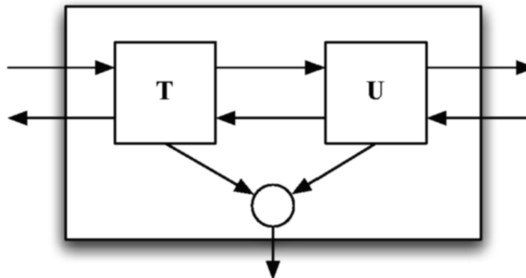



**Figure 6.** Sequential Composition: ***T* ; *U***

This definition of sequential composition is associative and has ***succeed*** as its unit. A simple proof of associativity is obtained by drawing the diagram for a sequential composition with three components, and adding an extra box, either around the two left operands or around the two right operands. It is easy to see that this represents the two different bracketings of the associative law. The flowchart itself remains the same in both cases.

The definition of sequential composition states that failure of any component transaction of the sequence will propagate inexorably to the left, until everything that has ever been done since the beginning of time has been undone. This is not always desirable. The ***else*** operator shown in Figure 7 gives a way of halting the stream of failures and compensations. It reverses again the direction of travel of the token, and tries a different way of achieving the same eventual goal.

At most one of these alternatives will actually take effect. The first of them is tried first. If it fails (having compensated of course), the second one is started. If this now succeeds, control is passed to the following transaction, so that it too may try again. As a result, the *finish* exit of the whole composition may be activated twice, or even more often if either of the alternatives itself finishes many times.

Note the fan-in at the *finish* exit: from the outside it is impossible to distinguish which alternative has succeeded on each occasion. Note also the fan-out of the *failback* arrow. In spite of this fan-out, the ***else*** construction is deterministic. When *failback* occurs, control may pass to the failback of either of the alternatives. The selection of destination will always be determined by the behavioural constraint on the component boxes. As a result, control will pass to the alternative that has most recently finished, which is obviously the right one to perform the appropriate compensation.

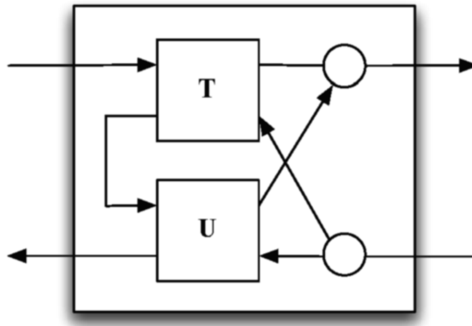



**Figure 7.** Alternative Composition: ***T else U***

In the uncertain world in which computers operate, especially in the vicinity of people, it is quite possible that a transaction that has failed once may succeed when it is simply tried again. But clearly the programmer should control how often to repeat the attempt. For example, suppose it is known that the transaction **U** is strictly compensable. Then the transaction

***(succeed else succeed else succeed) ; U***

merely causes ***U*** to be repeated up to three times – that is, up to three times more often than this transaction itself is repeated by its own predecessors.

The ***else*** operator is associative with unit ***fail***. The proof is no more difficult than that for sequential composition.

Because of its deterministic order of execution, the ***else*** command is asymmetric. Sometimes the programmer does not care which choice is made, and it is acceptable to delegate the choice to the implementation. For this reason, we introduce an ***or*** constructor, which is symmetric but non-deterministic. Its pictorial definition is very regular, but too cluttered to be worth drawing explicitly. Each entry of the outer box fans out to the like-named entry of both inner boxes. Each exit from the outer box fans in from the like-named exits of the two inner boxes. The non-determinism is introduced by the fan-out of the *start* arrow, which leads to two entries that are both ready to accept the token. After the start, the behavioural constraint ensures that the rejected alternative will never obtain the token. Note that the *fail* arrow of the whole box fans in from the *fail* arrows of both its operands. This shows that the whole box may fail if **either** of its two operands fails. In this respect, non-determinism differs from (and is worse than) the **else** construction, which guarantees to recover from any single failure.

There is yet a third form of choice between alternatives, which plays the role of the external choice in a process algebra. It is denoted by [] in CSP or + in CCS. External choice is slightly more deterministic than ***or***, and a bit less deterministic than ***else***. Like ***or*** it is symmetric. Like ***else*** it recovers from any single failure. It is defined by means of an ***else***, where the order of trying the operands is non-deterministic.

***T [] U = ( T else U ) or ( U else T )***

A picture of this operator would have to contain two copies of each of the operands; it is not worth drawing. A conventional equational definition is to be preferred.

This construction ***T [] U***

- fails if both ***U*** and ***T*** fail
- does ***U*** if ***T*** fails
- does ***T*** if ***U*** fails
- chooses non-deterministically if neither fails
- may throw if either ***T*** or ***U*** can do so.

A ***catch*** is similar to an ***else*** in providing an alternative way of achieving the same goal. The difference is that the first operand does not necessarily restore its initial state, and that the second operand is triggered by a *throw* exit instead of a *fail* exit from the first operand. A throw is appropriate when the first operand has been unable either to restore the initial state or to finish successfully. The catching clause is intended to behave like the first operand should have done: either to complete the compensation and fail, or to succeed in the normal way, or else to throw again to some yet more distant catch. Note that the catching clause does not satisfy the assertional constraint for a compensable transaction, because the assertion at its *start* is not the same as the assertion at its *fail* exit.

## 5. Nested Transactions

We have described in the previous section how a primitive transaction can be declared by specifying a forward action together with its compensation. In the elementary case, both of these are ordinary sequential programs. In this section we will also allow the forward action to be itself a long-running transaction (which we call the child transaction), nested inside a larger parent transaction declaration, as shown in Figure 8. As before, the compensation **U** of the parent transaction is an ordinary sequential program, and is triggered from the *failback* entry of the parent transaction. As a result, the *failback* entry of the child transaction **T** is never activated. As a result, when the parent transaction is complete, an implementation can discard the accumulated child compensations, and recover the stack frames and other declared resources of the child transactions.

Nested transactions can be useful as follows. When a long sequence of transactions all succeed, they build up a long sequence of compensations to be executed (in reverse order) in the event of subsequent failure. However, at a certain stage there may be some much better way of achieving the compensation, as it were in a single big step right back to the beginning, rather than in the sequence of small steps accumulated by the child transactions. The new single-step compensation is declared as the compensation for the parent transaction. An example can be taken from a word processing program, where each child transaction deals with a single keystroke, and undoes it when required to compensate. However, when the parent document is complete, any subsequent failure will be compensated by restoring the previous version of the whole document.

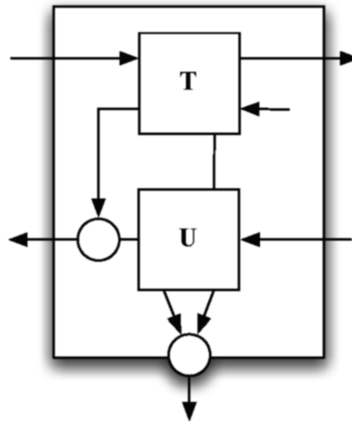



**Figure 8.** Nested Transaction Declaration

When the child transactions have all finished, their *failback* entries will never subsequently be activated, because (when necessary) the parent compensation is called instead. As a result, at the *finish* of the parent transaction an implementation can simply discard the accumulated child compensations, and recover the stack frames that they occupied.

In addition to stack frames, there may be other resources which need to be released by the child transactions on completion of the parent transaction. In the case of failure, the compensation can do this. But if all the child transactions succeed, we need another

mechanism. To provide this requires a significant extension to our definition of a transaction. We add to every transaction (the child transactions as well as the parent) a new entry point called *finally*, placed between the *start* and the *fail*, and a new exit point called *complete*, placed between the *finish* and the *failback*. The nestable transaction declaration therefore takes a third operand, a *completion* action; it is entered by the *finally* entry and exited by the *complete* exit.

When transactions (parents or children) are composed sequentially, their completions are also composed sequentially, like their compensations, by connecting the *complete* exit of the left operand to the *finally* entry of the right operand. So the connecting arrows between completions go from left to right, and the completions are executed in the same order as the forward actions, rather than in the reverse order.

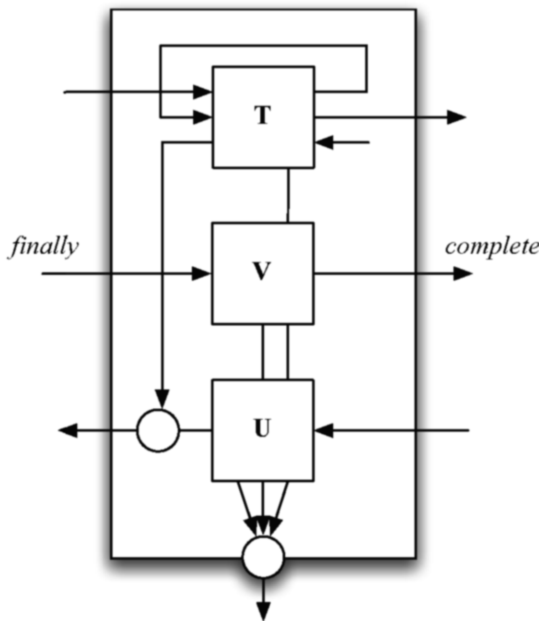



**Figure 9.** Nesting with Completion: **[ T finally V comp U ]**

In Figure 9, the child transaction is denoted ***T*** , the parent compensation is ***U***, and the parent completion is ***V***. The child transaction also has a *finally* entry and a *complete* exit, and a completion action, which is not shown explicitly in the diagram. In the case that the child is not a transaction, an ordinary sequential program can be artificially made into a transaction by adding a completion action that does nothing. In that case, the definition of a nested transaction becomes equivalent to that of an un-nested one.

When the whole child transaction has finished, the completions accumulated by the child transactions are triggered. That is indicated in Figure 9 by the transverse arrow from the *finally* exit of the child transactions ***T*** to the new *finally* entry of the child transactions themselves. It executes the completion actions of all the children, in the same order as the execution of forward actions of the children.

Another benefit of the introduction of completion actions is to implement the *lazy update* design pattern. Each child makes a generally accessible note of the update that it is responsible for performing, but lazily does not perform the update until all the child transactions of the same parent have successfully finished. On seeing the note, the forward action of each subsequent child takes account of the notes left by all previously executed children, and behaves as if the updates postponed by all previous children had already occurred. But on *completion* of the parent transaction, the real updates are actually performed by the completion code provided by each component child transaction. As a result, the rest of the world will never know how lazy the transaction has been. The completion codes will be executed in the same sequence as the forward actions of the children. Compensations for lazy transactions tend to be rather simple, since all that is required is to throw away the notes on the actions that should have been performed but have not yet been.

Introduction of the new *finally* entry and *complete* exit for completion actions requires an extension to the definition of the behavioural constraint on transactions. Note that a completion is not allowed to fail, though it may still throw.

*start ; X*

where

*X = fail + throw + (finish ; ( finally ; ( complete + throw ) + failback ; X ))*

The definition of sequential composition and other operators needs to be adapted to accommodate the addition of new entry and exit points for the completion actions. The adaptations are fairly obvious, and we leave them to the interested reader.

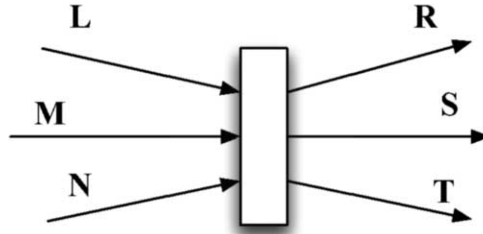
The nesting of transactions may seem unpleasantly complex, but the concept of nesting is essential to deal with the wide range of granularity at which the concept of atomicity can be applied. Many kinds of transaction will last a few microseconds, whereas others may last a few months.

## 6. Concurrency

The Petri net place has provided a mechanism for fan-in and fan-out of the arrows of a flowchart. Each activation (firing) of the place involves the entry of a single token along a single **one** of the entry arrow, and the exit of the same token along any **one** of its exit arrows. As a result, a place always maintains the number of tokens in the net – in our treatment so far, there has only been just one token.

Introduction and elimination of tokens from the net is the purpose of the other primitive element of a Petri net, the transition. This too provides a form of fan-in and fan-out, but its behavioural rule is conjunctive rather than disjunctive, universal rather than existential. Each firing of a transition requires the entry of a token on **all** of its entry arrows, and the emission of a token on **all** of its exit arrows. The notation used for transitions is shown in Figure 10.

If there is only one entry to a transition, it acts as a fan-out: its firing will increase the number of tokens travelling simultaneously in the network. This could certainly lead to confusion if one of the tokens ever meets another at the same place. By allowing only limited and well-structured forms of composition, our calculus will confine each token to a disjoint region of the net, and ensure that tokens meet only at the entry to a transition,




Correctness condition: **L & M & N ⇒ R & S & T**
Behaviour: $(l||m||n);\ (r||s||t)$

**Figure 10.** Petri net transition

which is what is intended. Often, such a meeting place is a fan-in; it has only one exit, so that it reduces the number of tokens in the system.

It is possible to think of all the entry and exit events for a transition as occurring simultaneously. However, in representing this simultaneous behaviour as a regular expression, it is common to use a total ordering of events, in which any causal event occurs before its effect. Furthermore, arbitrary interleaving is commonly used to record sequentially events that occur simultaneously. The regular expression (P || Q) will stand for the set of all interleavings of a string from P with a string from Q . Thus the behavioural constraint on a transition in Figure 10 is that the arrival in any order of a token on all of the entry arrows will trigger the emission of a token on each and every one of the exit arrows, again in any order.

The correctness of a transition obviously requires that **all** the assertions on all the exit arrows must be valid at the time of firing. In this respect, the transition is like a place. It differs from a place in the precondition that **all** the assertions on the entry arrows may be assumed to be true when the transition fires. Thus the correctness condition on a transition is that the conjunction of all the entry assertions must logically imply the conjunction of all the exit assertions. In general, there is a possibility that the conjunction will be inconsistent; but we will design our calculus carefully to avoid this risk.

The semantics of the Petri net transition is given in terms of its correctness condition and its behaviour. Thus it satisfies the same equivalence criterion as the place: any acyclic network of pure transitions (in which every external exit is reachable from every external entry) is equivalent to a single transition with exactly the same external entry and exit arrows, but omitting the internal arrows.

We will explain the concept of well-structured concurrency first in the context of ordinary programs, which have only a single entry and a single exit arrow. Concurrent composition of two such programs is made to satisfy the same constraint, as shown in Figure 11. This shows how two sections of code **T** and **U** will start simultaneously and proceed concurrently until they have both finished. Only then does the concurrent combination finish.

It is evident from the diagram (and from the structured property of boxes) that the only meeting point of the two tokens generated by the fan-out transition on the left will be at the final fan-in transition on the right, where they are merged. The diagram can easily be adapted to deal with three or more threads. But this is not necessary, because

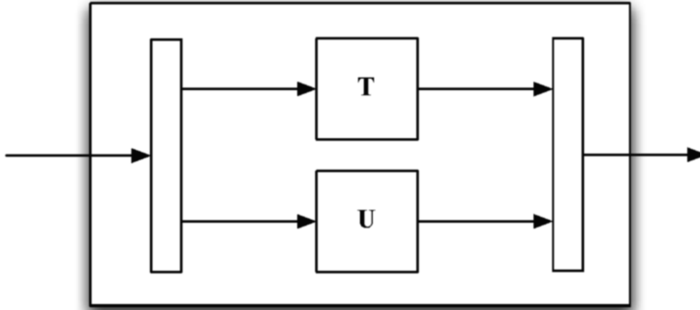



**Figure 11.** Parallel composition: **T || U**

the rules of equivalence for transitions ensure that concurrent composition is both an associative and a commutative operator.

The proof of correctness of concurrent threads should be modular in the same way as proof of correctness of all the other forms of composition. In order to make this possible, some disjointness constraints must be placed on the actions of the individual threads. The simplest constraint is that no thread can access any variable updated by some other concurrent thread. This same constraint must also be applied to the assertions used to prove correctness of each thread. The token which travels within a thread can be regarded as carrying the variables and owned by that thread, together with their values.

In simple cases, the constraint on disjointness can be enforced by a compile-time check on the global variables accessed by a thread. But in general, the use of indirect addressing (for example, in an object-oriented program) will make it necessary to prove disjointness by including some notion of ownership into the assertion language. Separation logic provides an elegant and flexible means of expressing disjointness of ownership, and establishing it by means of proof. However, we will not pursue this issue further here.

The disjointness constraint is effective in ensuring consistency of the final assertions of the threads when they all terminate together. It also avoids race conditions at run time, and so prevents any form of unwanted interference between the activities of the threads. However, it also rules out any form of beneficial interaction or cooperation between them. In particular, it rules out any sharing of internal storage or communication channels. A safe relaxation of this restriction is provided by atomic regions (or critical sections). This is defined as a section of code inside a thread, which is allowed to access and update a shared resource. The implementation must guarantee (for example by an exclusion semaphore) that only one thread at a time can be executing inside an atomic region, so race conditions are still avoided. The overall effect of multiple threads updating the shared resource includes an arbitrary interleaving of the execution of their complete atomic regions.

The Petri net formalisation of an atomic region models a shared resource as a token, which may be regarded as carrying the current state and value of the resource. At the beginning of an atomic region, a thread acquires ownership of this token in order to access and update the shared resource; and at the end of the region the shared resource is

released. Of course, execution of the region also requires the normal sequential token of the thread that contains it.

An atomic region is defined (Figure 12) as a sort of inverse of concurrent composition, with a fan-in at the beginning and a fan-out at the end. For simplicity, we assume that there is only a single shared resource, consisting of everything except the private resources of the currently active individual threads. In most practical applications, many separate resources will need to be declared, but we shall not deal with that complexity here.

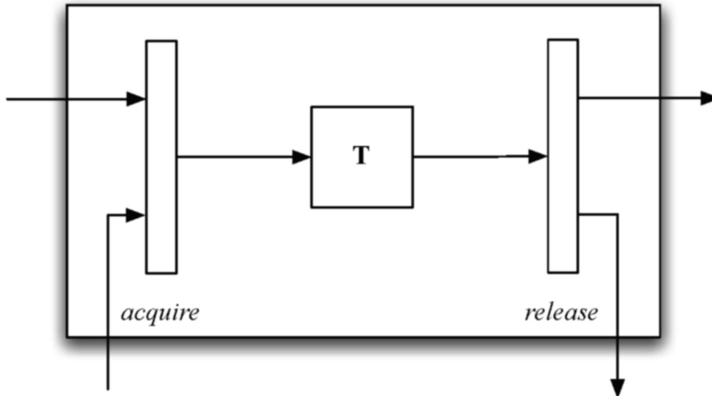



**Figure 12.** Atomic Region: **atomic[T]**

The definition of an atomic region requires the introduction of another entry and another exit into the standard repertoire. The entry carries the suggestive name *acquire* , and the exit is called *release*. The new entries and exits require extension of the behavioural constraint, by inserting (*acquire*;*release*)* between every entry and the next following exit. The definition of all the operators of our calculus must also be extended: but this is very simple, because in each diagram defining an operator, all the *acquire* entries are connected via a fan-out place, and all the *release* exits are connected via a fan-in place.

The declaration of a sharable resource is shown in Figure 13. The token that represents a resource is created by means of a transition fan-out. The block **T** contains all the multiple threads that are going to share the resource. When all of them have finished, the token is therefore merged again. The assertion **R** is known as the resource invariant: it must be true at the beginning of **T** and at the end of every atomic region within **T**. Conversely, **R** may be assumed true at the end of the whole block **T** , and at the beginning of every atomic region within it. Note that in this diagram the place is expected to store the token between successive executions of the atomic regions.

The explicit statement of a resource invariant permits a very necessary relaxation of the restriction that the assertions used within the threads of **T** may not refer to the values of the variables of the shared resource, for fear that they are subject to concurrent update by concurrent threads. The relaxed restriction states that all of the assertions private to a thread (initial, internal or final) may mention the values of the shared resource, but only in a way that is tolerant of interference. This means that the local assertions of each

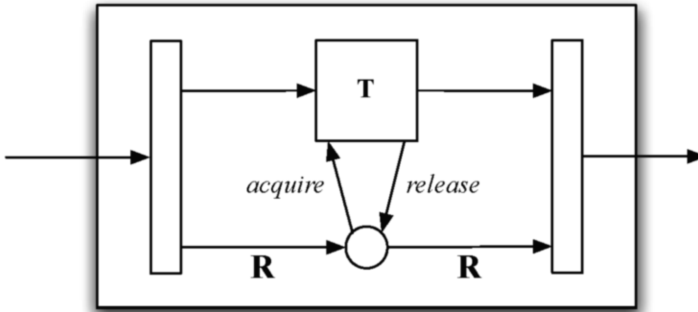



**Figure 13.** Resource Declaration: **resource R in T**

thread must also be an invariant of every atomic region that may be invoked by the other threads.

A direct application of this proof rule would require proof of each thread to know all the internal assertions in every other thread – a serious violation of the principal of locality of proof. A stronger but more modular condition is that each thread must prove locally that all its assertions are invariant of any section of code **X** that leaves **R** invariant. The details of the formalisation will not be elaborated here.

The definition of concurrent composition given in Figure 11 applies to fragments of ordinary program with a single entry and a single exit. Our final task is to apply the same idea to compensable transactions, with many more entries and exits. The basic definition of concurrency of transactions introduces a transition to fan out each entry of the concurrent block to the two (or more) like-named entries of the components; and similarly, it introduces a transition to fan in the like-named exits of the components to relevant exit of the whole composition (Figure 14). This means that the compensations of concurrent transactions will also be executed concurrently in the same way as their forward actions.

This scheme works well, provided that both components agree on which exit to activate on each occasion – either they both finish, or they both fail, or they both throw. The availability of a shared resource enables them to negotiate an agreement as required. However, if they fail to do so, the result is deadlock, and no further action is possible. It may be a good idea to complicate the definition of concurrency of transactions to avoid this unfortunate possibility automatically, by doing something sensible in each case. Four additional transitions are needed to implement the necessary logic.

1. if one component **T** finishes and **U** fails, these two exits are fanned in by a transition, whose exit leads to the *failback* of the successful **T**.
2. Similarly, if **U** finishes and **T** fails, the *failback* of **U** is invoked.
3. In the case that **T** performs a throw but **U** does not, the whole construction must throw. This involves connecting **U**'s **finish** and **fail** exits via a place to a transition that joins it with the *throw* exit of **T**.
4. A similar treatment deals with the case that **U** performs a *throw*.

Figure 15 shows the network controlling activation of the throw exit of **[ T || U ]**. It is easy to calculate that the correctness condition of the network is

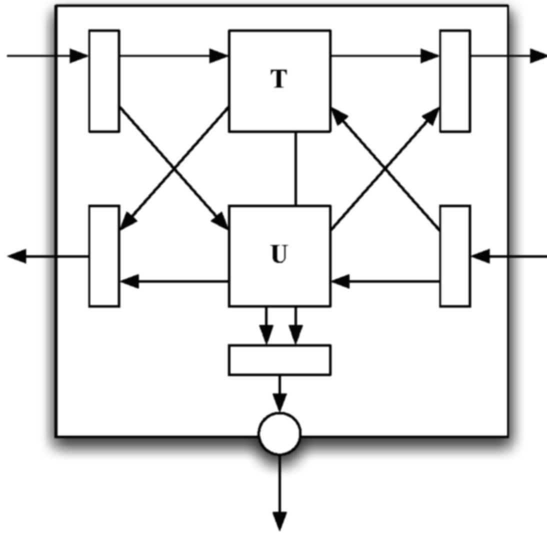



**Figure 14.** **[ T || U ]** as a transaction

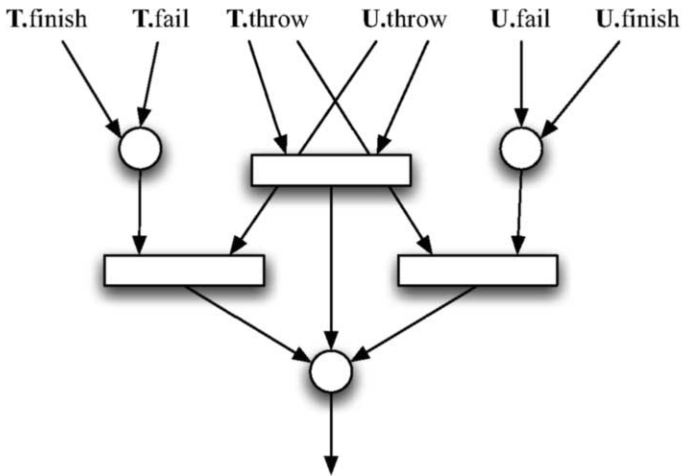



**Figure 15.** Network for *throw*

$$(\mathbf{T}.\mathit{finish} \vee \mathbf{T}.\mathit{fail}) \wedge \mathbf{T}.\mathit{throw} \vee \mathbf{T}.\mathit{throw} \wedge \mathbf{U}.\mathit{throw} \vee \mathbf{U}.\mathit{throw} \wedge (\mathbf{U}.\mathit{fail} \vee \mathbf{U}.\mathit{finish})$$

## 7. Conclusion

This paper gives a simple account using Petri nets of long-running transactions with compensations. The account is also quite formal, in the sense that the nets for any trans-

action composed solely by the principles described can actually be drawn, programmed and executed by computer. The assertions on the arrows give guidance on how to design correctness into a system of transactions from the earliest stage. The correctness principle for places and transitions serves as an axiomatic semantics, and shows how to prove the correctness of a complete flowchart in a modular way, by proving the correctness of each component and each connecting arrow separately. Thus we have given a unified treatment of both an operational and an axiomatic semantics for compensable, composable and nestable transactions. Simple cases of concurrency can also be treated, but more work, both theoretical and experimental, is needed to deal with more general cases.

The more surprising ideas in this article are (1) use of the precondition of a transaction as the criterion of adequacy of an approximation to the initial state that the compensation should reach (there are many more complicated ways of doing this); and (2) the suggestion of separation logic as an appropriate language for annotating the transitions of a concurrent Petri net.

The deficiencies of this article are numerous and obvious. There are no transition rules, no deductive systems, no algebraic axioms, no denotational semantic functions, no proofs, no examples and no references. There is far more work still to be done by anyone sufficiently interested in the subject.

## Acknowledgements

The ideas and presentation of this paper have been greatly improved by the helpful comments of:

Michael Butler, Ernie Cohen, Tim Harris, Niels Lohman, Jay Misra, Eliot Moss, Matthew Parkinson, Simon Peyton Jones, Viktor Vafeiadis.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Automata on Infinite Words and Their Applications in Formal Verification

Orna KUPFERMAN

*School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel*

**Abstract.** In *formal verification*, we check the correctness of a system with respect to a desired property by checking whether a mathematical model of the system satisfies a specification that formally expresses the property. In the *automata-theoretic approach* to formal verification, we model both the system and the specification by automata. Questions about systems and their specifications are then reduced to questions about automata. The goal of this course is to teach the basics of automata on infinite words and their applications in formal verification.

**Keywords.** Automata on infinite words, Formal verification.

## Introduction

Finite automata on infinite objects were first introduced in the 60's, and were the key to the solution of several fundamental decision problems in mathematics and logic [Büc62,McN66,Rab69]. Today, automata on infinite objects are used for specification and verification of nonterminating programs [Kur94,VW94]. The idea is simple: when a system is defined with respect to a finite set $P$ of propositions, each of the system's states can be associated with a set of propositions that hold in this state. Then, each of the system's computations induces an infinite word over the alphabet $2^P$, and the system itself induces a language of infinite words over this alphabet. This language can be defined by an automaton. Similarly, a specification for a system, which describes all the allowed computations, can be viewed as a language of infinite words over $2^P$, and can therefore be defined by an automaton. In the automata-theoretic approach to verification, we reduce questions about systems and their specifications to questions about automata. More specifically, questions such as satisfiability of specifications and correctness of systems with respect to their specifications are reduced to questions such as nonemptiness and language containment. The automata-theoretic approach separates the logical and the combinatorial aspects of reasoning about systems. The translation of specifications to automata handles the logic and shifts all the combinatorial difficulties to automata-theoretic problems. We will define automata on infinite words, study some of their properties, and see how they are used in formal verification.

## 1. The temporal logic LTL

The logic *LTL* is a linear temporal logic [Pnu77]. Formulas of LTL are constructed from a set $AP$ of atomic propositions using the usual Boolean operators and the temporal operators $X$ ("next time") and $U$ ("until"). Formally, an LTL formula over $AP$ is defined as follows:

- **true**, **false**, or $p$, for $p \in AP$.
- $\neg\psi_1$, $\psi_1 \wedge \psi_2$, $X\psi_1$, or $\psi_1 U \psi_2$, where $\psi_1$ and $\psi_2$ are LTL formulas.

We define the semantics of LTL with respect to an infinite *computation* $\pi = \sigma_0, \sigma_1, \sigma_2, \ldots$, where for every $j \geq 0$, the set $\sigma_j \subseteq AP$ is the set of atomic propositions that hold in the $j$-th position of $\pi$. We denote the suffix $\sigma_j, \sigma_{j+1}, \ldots$ of $\pi$ by $\pi^j$. We use $\pi \models \psi$ to indicate that an LTL formula $\psi$ holds in the computation $\pi$. The relation $\models$ is inductively defined as follows:

- For all $\pi$, we have that $\pi \models \mathbf{true}$ and $\pi \not\models \mathbf{false}$.
- For an atomic proposition $p \in AP$, we have that $\pi \models p$ iff $p \in \sigma_0$.
- $\pi \models \neg\psi_1$ iff $\pi \not\models \psi_1$.
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$.
- $\pi \models X\psi_1$ iff $\pi^1 \models \psi_1$.
- $\pi \models \psi_1 U \psi_2$ iff there exists $k \geq 0$ such that $\pi^k \models \psi_2$ and $\pi^i \models \psi_1$ for all $0 \leq i < k$.

We denote the size of an LTL formula $\varphi$ by $|\varphi|$ and we use the following abbreviations in writing formulas:

- $\vee$, $\rightarrow$, and $\leftrightarrow$, interpreted in the usual way.
- $F\psi = \mathbf{true} U \psi$ ("eventually", and the "F" comes from "Future").
- $G\psi = \neg F \neg \psi$ ("always", and the "G" comes from "Globally").

**Example 1.1** We specify in LTL some properties that one may wish a mutual exclusion algorithm to satisfy.

- The *mutual exclusion* property states that two processes are never simultaneously in their critical sections. If $c_i$ is an atomic proposition that hold when process $i$ is in its critical section, then the LTL formula $\psi_{me}^{i,j} = G(\neg c_i \vee \neg c_j)$ expresses mutual exclusion between processes $i$ and $j$. Note that the formula $\neg F(c_i \wedge c_j)$ is equivalent to $\psi_{me}^{i,j}$.
- The *finite waiting* property for process $i$ states that if process $i$ tries to access its critical section, it will eventually access it. If for each process $i$, the atomic proposition $t_i$ holds when process $i$ tries to enter the critical section, then the LTL formula $\psi_{fw}^{i} = G(t_i \rightarrow F c_i)$ expresses finite waiting for process $i$. Note that the semantics of the operator $U$ (and therefore also the one of $F$) includes the present in the future. Thus, the requirement to have $c_i$ eventually is satisfied if the process $i$ is already in the critical section when it tries. A different specification is $G(t_i \rightarrow XFc_i)$, in which the access of the critical section has to be in the strict future.

We interpret LTL formulas also with respect to *Kripke structures*, which may generate many computations. Formally, a Kripke structure is $K = \langle AP, W, R, W_0, L\rangle$, where

$W$ is a set of states, $R \subseteq W \times W$ is a total transition relation (that is, for every $w \in W$, there is at least one $w'$ such that $R(w, w')$, the set $W_0 \subseteq W$ is a set of initial states, and $L : W \rightarrow 2^{AP}$ maps each state to the sets of atomic propositions that hold in it. A path of $K$ is an infinite sequence $w_0, w_1, \ldots$ such that $w_0 \in W_0$ and for all $i \geq 0$ we have $R(w_i, w_{i+1})$. Every path $w_0, w_1, \ldots$ of $K$ induces the computation $L(w_0), L(w_1), \ldots$ of $K$.

The *model-checking problem* for LTL is to determine, given an LTL formula $\psi$ and a Kripke structure $K$, whether all the computations of $K$ satisfy $\psi$ [CE81,QS81,LP85, VW94].

## 2. Büchi word automata

We can view Kripke structures as generators of languages over the alphabet $2^{AP}$. We can also view properties as descriptions of languages over this alphabet.

**Example 2.1** The properties specified by LTL formulas in Example 1.1, corresponds to the following languages over the alphabet $2^{AP}$.

- The language $L^{i,j}_{me}$ that corresponds to mutual exclusion contains all computations having no occurrences of letters containing both $c_i$ and $c_j$. Formally,

$$L^{i,j}_{me} = \{\sigma_0 \cdot \sigma_1 \cdots : \text{ for all } l \geq 0, \text{ we have } c_i \notin \sigma_l \vee c_j \notin \sigma_l\}.$$

- The language $L^{i}_{fw}$ that corresponds to finite waiting for process $i$ contains all computations in which every occurrence of a letter containing $t_i$ is followed by an occurrence of a letter containing $c_i$.

$$L^{i}_{fw} = \{\sigma_0 \cdot \sigma_1 \cdots : \text{ for all } l \geq 0, \text{ if } t_i \in \sigma_l, \text{ then there is } k \geq l \text{ with } c_i \in \sigma_k)\}.$$

We describe and reason about languages of infinite words using automata on infinite words. Let $\Sigma$ be a finite alphabet. A *Büchi word automaton* is $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, where $\Sigma$ is the input alphabet, $Q$ is a finite set of states, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $\alpha \subseteq Q$ is a set of accepting states. Since $\mathcal{A}$ may have several initial states and since the transition function may specify many possible transitions for each state and letter, $\mathcal{A}$ may be *nondeterministic*. If $|Q_0| = 1$ and $\delta$ is such that for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\delta(q, \sigma)| \leq 1$, then $\mathcal{A}$ is a *deterministic* automaton. We use NBW and DBW as abbreviations for nondeterministic Büchi word automata and deterministic Büchi word automata, respectively.

Given an input word $w = \sigma_0 \cdot \sigma_1 \cdots$ in $\Sigma^\omega$, a *run* of $\mathcal{A}$ on $w$ is a function $r : \mathbb{N} \rightarrow Q$ where $r(0) \in Q_0$ and for every $i \geq 0$, we have $r(i+1) \in \delta(r(i), \sigma_i)$; i.e., the run starts in one of the initial states and obeys the transition function. Note that a nondeterministic automaton can have many runs on $w$. In contrast, a deterministic automaton has a single run on $w$. For a run $r$, let $inf(r)$ denote the set of states that $r$ visits infinitely often. That is,

$$inf(r) = \{q \in Q \ : r(i) = q \text{ for infinitely many } i \geq 0\}.$$

As $Q$ is finite, it is guaranteed that $inf(r) \neq \emptyset$. The run $r$ is *accepting* iff $inf(r) \cap \alpha \neq \emptyset$. That is, iff there exists a state in $\alpha$ that $r$ visits infinitely often. A run that is not accepting is *rejecting*. An automaton $\mathcal{A}$ accepts an input word $w$ iff there exists an accepting run of $\mathcal{A}$ on $w$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts.

**Example 2.2** In Figure 1 we describe two Büchi automata. The alphabet of both automata is $\{a, b\}$. The automaton $\mathcal{A}_1$, which is deterministic, accepts exactly all words with infinitely many $a$'s. The automaton $\mathcal{A}_2$ complements it and accepts exactly all words with finitely many $a$'s.

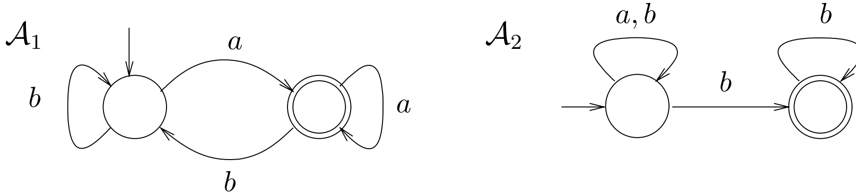



**Figure 1.** An example of nondeterministic Büchi automata.

The Büchi acceptance condition suggests one way to refer to $inf(r)$ in order to determine whether the run $r$ is accepting. More acceptance conditions are defined in the literature. Below we define a generalization of the Büchi condition. A *generalized Büchi automaton* is $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, where $\Sigma$, $Q$, $\delta$, and $Q_0$ are as in Büchi automata, and the acceptance condition $\alpha \subseteq 2^Q$ consists of sets $\alpha_i \subseteq Q$. A run $r$ of $\mathcal{A}$ with $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ is accepting iff $inf(r) \cap \alpha_i \neq \emptyset$ for all $1 \leq i \leq k$. That is, $r$ is accepting if every set in $\alpha$ is visited infinitely often. We refer to $k$ as the *index* of the generalized Büchi automaton.

In Question 4, you will prove that generalized Büchi automata are not more expressive than Büchi automata: given a nondeterministic generalized Büchi automaton with $n$ states and index $k$, it is possible to construct an equivalent Büchi automaton with $nk$ states.

## 3. Properties of Büchi Automata

It is easy to see that büchi automata are closed under union. Below we prove closure under intersection.

**Theorem 3.1** [Cho74] *Given Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we can construct a Büchi automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.*

**Proof:** Let $\mathcal{A}_1 = (\Sigma, Q_1, Q_1^0, \delta_1, \alpha_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, Q_2^0, \delta_2, \alpha_2)$. We define $\mathcal{A} = \langle \Sigma, Q, Q^0, \delta, \alpha \rangle$, where

- $Q = Q_1 \times Q_2 \times \{1, 2\}$,
- $Q^0 = Q_1^0 \times Q_2^0 \times \{1\}$,

- $\delta(\langle s,t,i\rangle,a) = \delta_i(s,a) \times \delta_2(t,a) \times \{j\}$, where $i = j$ unless $i = 1$ and $s \in \alpha_1$, in which case $j = 2$, or $i = 2$ and $t \in \alpha_2$, in which case $j = 1$, and
- $\alpha = \alpha_1 \times Q_2 \times \{1\}$.

The automaton $\mathcal{A}$ has two copies of the product of $\mathcal{A}_1$ and $\mathcal{A}_2$. When executing a run, if we are in the first copy (the second copy), and we visit an accepting state of $\mathcal{A}_1$ ($\mathcal{A}_2$), we move to the other copy. The accepting condition requires to go infinitely often through $\alpha_1 \times Q_2 \times \{1\}$, i.e., to visit infinitely often accepting states of $\mathcal{A}_1$ in the first copy. Since we can return to the first copy only after having visited an accepting state of $\mathcal{A}_2$ on the second copy, both copies visit accepting states infinitely often. □

We now turn to study the expressive power of nondeterministic vs. deterministic Büchi automata. Recall that in the case of finite words, automata can be determinized, thus nondeterministic automata on finite words are not more expressive than deterministic automata on finite words. We now show that this is not the case for the infinite-word setting.

**Theorem 3.2** [Lan69] *Deterministic Büchi automata are strictly less expressive than nondeterministic Büchi automata.*

**Proof:** Consider the language $L = (a+b)^* b^\omega$, (i.e., $L$ consists of all infinite words in which $a$ occurs only finitely many times). As shown in Example 2.2, the language $L$ is recognizable by a nondeterministic Büchi automata. We now show that $L$ is not recognizable by a deterministic Büchi automaton. Assume by way of contradiction that $L = \mathcal{L}(A)$, for a deterministic $\mathcal{A} = (\{a,b\}, Q, \{q_0\}, \delta, \alpha)$. Recall that $\delta$ can be viewed as a partial mapping from $Q \times \{a,b\}^*$ to $Q$.

Consider the infinite word $w_0 = b^\omega$. Clearly, $w_0$ is accepted by $\mathcal{A}$, so $\mathcal{A}$ has an accepting run on $w_0$. Thus, $w_0$ has a finite prefix $u_0$ such that $\delta(q_0, u_0) \in \alpha$. Consider now the infinite word $w_1 = u_0 a b^\omega$. Clearly, $w_1$ is also accepted by $\mathcal{A}$, so $\mathcal{A}$ has an accepting run on $w_1$. Thus, $w_1$ has a finite prefix $u_0 b u_1$ such that $\delta(q_0, u_0 a u_1) \in \alpha$. In a similar way we can continue to find finite words $u_i$ such that $\delta(q_0, u_0 a u_1 a \ldots a u_i) \in \alpha$. Since $Q$ is finite, there are $i, j$, where $0 \leq i < j$, such that $\delta(q_0, u_0 a u_1 a \ldots a u_i) = \delta(q_0, u_0 a u_1 a \ldots a u_i a \ldots a u_j)$. It follows that $\mathcal{A}$ has an accepting run on

$$u_0 a u_1 a \ldots a u_i (a u_{i+1} \ldots u_{j-1} a u_j)^\omega.$$

But the latter word has infinitely many occurrences of $a$, so it is not in $L$. □

The *complement* of an NBW $\mathcal{A}$ is an NBW $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$. In the case of finite words, we complement automata by determinizing them and then dualizing the acceptance condition (that is, defining the set of accepting states to be $Q \setminus \alpha$). While it is possible to complement a DBW (but not by dualization, see Question 5), Theorem 3.2 implies we cannot use such a complementation as an intermediate step in general NBW complementation. We will get back to the complementation problem for NBW in Section 7.

## 4. From LTL to NBW

Given an LTL formula $\psi$, we construct a generalized Büchi word automaton $\mathcal{A}_\psi$ such that $\mathcal{A}_\psi$ accepts exactly all the computations that satisfy $\psi$. The construction was first suggested by Vardi and Wolper in 1986.

For an LTL formula $\psi$, the *closure* of $\psi$, denoted $cl(\psi)$, is the set of $\psi$'s subformulas and their negation ($\neg\neg\psi$ is identified with $\psi$). Formally, $cl(\psi)$ is the smallest set of formulas that satisfy the following.

- $\psi \in cl(\psi)$.
- If $\psi_1 \in cl(\psi)$ then $\neg\psi_1 \in cl(\psi)$.
- If $\neg\psi_1 \in cl(\psi)$ then $\psi_1 \in cl(\psi)$.
- If $\psi_1 \wedge \psi_2 \in cl(\psi)$ then $\psi_1 \in cl(\psi)$ and $\psi_2 \in cl(\psi)$.
- If $X\psi_1 \in cl(\psi)$ then $\psi_1 \in cl(\psi)$.
- If $\psi_1 U \psi_2 \in cl(\psi)$ then $\psi_1 \in cl(\psi)$ and $\psi_2 \in cl(\psi)$.

For example, $cl(p \wedge ((Xp)Uq))$ is

$$\{p \wedge ((Xp)Uq), \neg(p \wedge ((Xp)Uq)), p, \neg p, (Xp)Uq, \neg((Xp)Uq), Xp, \neg Xp, q, \neg q\}.$$

**Theorem 4.1** [VW94] *Given an LTL formula $\psi$, we can construct an NBW $\mathcal{A}_\psi$ such that $\mathcal{L}(\mathcal{A}_\psi)$ is exactly the set of words satisfying $\psi$ and the size of $\mathcal{A}_\psi$ is exponential in the length of $\psi$.*

**Proof:** We define $\mathcal{A}_\psi = \langle 2^{AP}, Q, \delta, Q_0, \alpha\rangle$, where

- We say that a set $S \subseteq cl(\psi)$ is *good in* $cl(\psi)$ if $S$ is a maximal set of formulas in $cl(\psi)$ that does not have propositional inconsistency. Thus, $S$ satisfies the following conditions.

  1. For all $\psi_1 \in cl(\psi)$, we have $\psi_1 \in S$ iff $\neg\psi_1 \notin S$, and
  2. For all $\psi_1 \wedge \psi_2 \in cl(\psi)$, we have $\psi_1 \wedge \psi_2 \in S$ iff $\psi_1 \in S$ and $\psi_2 \in S$.

  The state space $Q \subseteq 2^{cl(\psi)}$ is the set of all the good sets in $cl(\psi)$.
- Let $S$ and $S'$ be two good sets in $cl(\psi)$, and let $\sigma \subseteq AP$ be a letter. Then $S' \in \delta(S, \sigma)$ if the following hold.

  1. $\sigma = S \cap AP$,
  2. For all $X\psi_1 \in cl(\psi)$, we have $X\psi_1 \in S$ iff $\psi_1 \in S'$, and
  3. For all $\psi_1 U \psi_2 \in cl(\psi)$, we have $\psi_1 U \psi_2 \in S$ iff either $\psi_2 \in S$ or both $\psi_1 \in S$ and $\psi_1 U \psi_2 \in S'$.

  Note that the last condition also means that for all $\neg(\psi_1 U \psi_2) \in cl(\psi)$, we have that $\neg(\psi_1 U \psi_2) \in S$ iff $\neg\psi_2 \in S$ and either $\neg\psi_1 \in S$ or $\neg(\psi_1 U \psi_2) \in S'$.
- $Q_0 \subseteq Q$ is the set of all states $S \in Q$ for which $\psi \in S$.
- Every formula $\psi_1 U \psi_2$ contributes to $\alpha$ the set

$$\alpha_{\psi_1 U \psi_2} = \{S \in Q : \psi_2 \in S \text{ or } \neg(\psi_1 U \psi_2) \in S\}.$$

□

## 5. Alternating Automata on Infinite Words

In Section 2 we defined Büchi automata and mentioned that automata on infinite words can be classified according to the type of acceptance condition. Another way to classify an automaton on infinite words is by the type of its branching mode. In a *deterministic* automaton, the transition function $\delta$ maps a pair of a state and a letter into a single state. The intuition is that when the automaton is in state $q$ and it reads a letter $\sigma$, then the automaton moves to state $\delta(q, \sigma)$, from which it should accept the suffix of the word. When the branching mode is existential, the automaton is nondeterministic and $\delta$ maps $q$ and $\sigma$ into a set of states. In the existential mode, the automaton should accept the suffix of the word from one of the states in the set. Accordingly, a word is accepted by a nondeterministic automaton if the automaton has some accepting run on it. We have met deterministic and nondeterministic automata in Section 2. In this section we meet more sophisticated branching modes. The *universal* branching mode is dual to the existential one. Thus, as there, $\delta$ maps $q$ and $\sigma$ into a set of states, yet the automaton should accept the suffix of the word from all of the states in the set. Accordingly, a word is accepted by a universal automaton if all the runs of the automaton on the word are accepting. In *alternating* automata [CKS81], both existential and universal modes are allowed, and the transitions are given as Boolean formulas over the set of states. For example, $\delta(q, \sigma) = q_1 \vee (q_2 \wedge q_3)$ means that the automaton should accept the suffix of the word either from state $q_1$ or from both states $q_2$ and $q_3$. We now define alternating automata formally and see how they can serve as an intermediate step in the translation of LTL to nondeterministic Büchi automata.

For a given set $X$, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over $X$ (i.e., Boolean formulas built from elements in $X$ using $\wedge$ and $\vee$), where we also allow the formulas $\mathbf{true}$ and $\mathbf{false}$. For $Y \subseteq X$, we say that $Y$ *satisfies* a formula $\theta \in \mathcal{B}^+(X)$ iff the truth assignment that assigns *true* to the members of $Y$ and assigns *false* to the members of $X \setminus Y$ satisfies $\theta$. For example, the sets $\{q_1, q_3\}$ and $\{q_2, q_3\}$ both satisfy the formula $(q_1 \vee q_2) \wedge q_3$, while the set $\{q_1, q_2\}$ does not satisfy this formula.

Consider an automaton $\mathcal{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$. For a word $w = \sigma_0 \cdot \sigma_1 \cdots$ and an index $i \geq 0$, let $w^i = \sigma_i \cdot \sigma_{i+1} \cdots$ be the suffix of $w$ that starts in position $i$. We can represent $\delta$ using $\mathcal{B}^+(Q)$. For example, a transition $\delta(q, \sigma) = \{q_1, q_2, q_3\}$ of a nondeterministic automaton $\mathcal{A}$ can be written as $\delta(q, \sigma) = q_1 \vee q_2 \vee q_3$. If $\mathcal{A}$ is universal, the transition can be written as $\delta(q, \sigma) = q_1 \wedge q_2 \wedge q_3$. While transitions of nondeterministic and universal automata correspond to disjunctions and conjunctions, respectively, transitions of alternating automata can be arbitrary formulas in $\mathcal{B}^+(Q)$. We can have, for instance, a transition $\delta(q, \sigma) = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$, meaning that the automaton accepts a suffix $w^i$ of $w$ from state $q$, if it accepts $w^{i+1}$ from both $q_1$ and $q_2$ or from both $q_3$ and $q_4$. Such a transition combines existential and universal choices.

Formally, an *alternating automaton on infinite words* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$, where $\Sigma, Q, q_{in}$, and $\alpha$ are as in nondeterministic automata (for technical simplicity we assume that the set of initial states is a singleton), and $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is a transition function. While a run of a nondeterministic automaton is an infinite sequence of states, a run of an alternating automaton is a tree $r : T_r \to Q$ for some $T_r \subseteq \mathbb{N}^*$. Formally, a tree is a (finite or infinite) nonempty prefix-closed set $T \subseteq \mathbb{N}^*$. The elements of $T$ are called *nodes*, and the empty word $\varepsilon$ is the *root* of $T$. For every $x \in T$, the nodes $x \cdot c \in T$ where $c \in \mathbb{N}$ are the *children* of $x$. A node with no children is a *leaf*. We sometimes refer to the

length $|x|$ of $x$ as its *level* in the tree. A *path* $\pi$ of a tree $T$ is a set $\pi \subseteq T$ such that $\epsilon \in \pi$ and for every $x \in \pi$, either $x$ is a leaf, or there exists a unique $c \in \mathbb{N}$ such that $x \cdot c \in \pi$. Given a finite set $\Sigma$, a $\Sigma$-*labeled tree* is a pair $\langle T, V \rangle$ where $T$ is a tree and $V : T \to \Sigma$ maps each node of $T$ to a letter in $\Sigma$. A run of $\mathcal{A}$ on an infinite word $w = \sigma_0 \cdot \sigma_1 \cdots$ is a $Q$-labeled tree $\langle T_r, r \rangle$ such that the following hold:

- $r(\varepsilon) = q_{in}$.
- Let $x \in T_r$ with $r(x) = q$ and $\delta(q, \sigma_{|x|}) = \theta$. There is a (possibly empty) set $S = \{q_1, \ldots, q_k\}$ such that $S$ satisfies $\theta$ and for all $1 \leq c \leq k$, we have $x \cdot c \in T_r$ and $r(x \cdot c) = q_c$.

For example, if $\delta(q_{in}, \sigma_0) = (q_1 \vee q_2) \wedge (q_3 \vee q_4)$, then possible runs of $\mathcal{A}$ on $w$ have a root labeled $q_{in}$, have one node in level 1 labeled $q_1$ or $q_2$, and have another node in level 1 labeled $q_3$ or $q_4$. Note that if $\theta = \mathbf{true}$, then $x$ need not have children. This is the reason why $T_r$ may have leaves. Also, since there exists no set $S$ as required for $\theta = \mathbf{false}$, we cannot have a run that takes a transition with $\theta = \mathbf{false}$.

A run $\langle T_r, r \rangle$ is *accepting* iff all its infinite paths, which are labeled by words in $Q^\omega$, satisfy the acceptance condition. A word $w$ is accepted iff there exists an accepting run on it. Note that while conjunctions in the transition function of $\mathcal{A}$ are reflected in branches of $\langle T_r, r \rangle$, disjunctions are reflected in the fact we can have many runs on the same word. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of infinite words that $\mathcal{A}$ accepts. We use ABW to abbreviate alternating Büchi automata on infinite words.

## 6. From LTL to NBW via alternating Büchi automata

In this section we show an alternative translation of LTL to NBW. The translation goes via alternating automata: we first translate the LTL formula to an ABW, and then translate the ABW to an an NBW. Using alternating automata as an intermediate step was first suggested in the branching framework, for the translation of branching temporal logics to tree automata [KVW00]. There, the use of alternating automata enables an efficient automata-based solution to the model checking problem. In the linear framework, an advantage of the intermediate alternating automaton is the ability to apply optimization algorithms on both the intermediate ABW and the final NBW [Fri03,FW02,GKSV03].

For simplicity, we assume that LTL formulas are in positive normal form, where negation is applied only to atomic propositions. Having no negation, we should have both $\wedge$ and $\vee$, and have the temporal operator $G$ in addition to $X$ and $U$.

**Theorem 6.1** [KVW00] *Given an LTL formula $\psi$ in positive normal form, we can construct an ABW $\mathcal{A}_\psi$ such that $\mathcal{L}(\mathcal{A}_\psi)$ is exactly the set of words satisfying $\psi$ and the size of $\mathcal{A}_\psi$ is linear in the length of $\psi$.*

**Proof:** We define $\mathcal{A}_\psi = \langle 2^{AP}, cl(\psi), \delta, \psi, \alpha \rangle$, as follows. The set $\alpha$ of accepting states consists of all the $G$-formulas in $cl(\psi)$; that is, formulas of the form $G\varphi_2$. The transition function $\delta$ is defined, for all $\sigma \in 2^{AP}$, as follows.

- $\delta(p, \sigma) = \mathbf{true}$ if $p \in \sigma$.
- $\delta(p, \sigma) = \mathbf{false}$ if $p \notin \sigma$.
- $\delta(\neg p, \sigma) = \mathbf{true}$ if $p \notin \sigma$.

- $\delta(\neg p, \sigma) = \mathbf{false}$ if $p \in \sigma$.
- $\delta(\varphi_1 \wedge \varphi_2, \sigma) = \delta(\varphi_1, \sigma) \wedge \delta(\varphi_2, \sigma)$.
- $\delta(\varphi_1 \vee \varphi_2, \sigma) = \delta(\varphi_1, \sigma) \vee \delta(\varphi_2, \sigma)$.
- $\delta(X\varphi_2, \sigma) = \varphi_2$.
- $\delta(\varphi_1 U \varphi_2, \sigma) = \delta(\varphi_2, \sigma) \vee (\delta(\varphi_1, \sigma) \wedge \varphi_1 U \varphi_2)$.
- $\delta(G\varphi_2, \sigma) = \delta(\varphi_2, \sigma) \wedge G\varphi_2$.

Intuitively, $\mathcal{A}_\psi$ follows the structure of the formula, and uses the acceptance condition $\alpha$ to guarantee that eventualities of $U$-formulas are eventually satisfied. □

In order to complete the translation of LTL to NBW, we need to remove alternation from $\mathcal{A}_\psi$:

**Theorem 6.2** [MH84] *Let $\mathcal{A}$ be an alternating Büchi automaton. There is a nondeterministic Büchi automaton $\mathcal{A}'$, with exponentially many states, such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.*

**Proof:** The automaton $\mathcal{A}'$ guesses a run of $\mathcal{A}$. At a given point of a run of $\mathcal{A}'$, it keeps in its memory a whole level of the run tree of $\mathcal{A}$. As it reads the next input letter, it guesses the next level of the run tree of $\mathcal{A}$. In order to make sure that every infinite path visits states in $\alpha$ infinitely often, $\mathcal{A}'$ keeps track of states that "owe" a visit to $\alpha$. Let $\mathcal{A} = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$. Then $\mathcal{A}' = \langle \Sigma, 2^Q \times 2^Q, \langle \{q_{in}\}, \emptyset \rangle, \delta', 2^Q \times \{\emptyset\} \rangle$, where $\delta'$ is defined, for all $\langle S, O \rangle \in 2^Q \times 2^Q$ and $\sigma \in \Sigma$, as follows.

- If $O \neq \emptyset$, then $\delta'(\langle S, O \rangle, \sigma) =$

$$\{\langle S', O' \setminus \alpha \rangle \mid S' \text{ satisfies } \bigwedge_{q \in S} \delta(q, \sigma), O' \subseteq S', \text{ and } O' \text{ satisfies } \bigwedge_{q \in O} \delta(q, \sigma)\}.$$

- If $O = \emptyset$, then $\delta'(\langle S, O \rangle, \sigma) =$

$$\{\langle S', S' \setminus \alpha \rangle \mid S' \text{ satisfies } \bigwedge_{q \in S} \delta(q, \sigma)\}.$$

□

In [MSS86], Muller et al. introduce *alternating weak automata*. In a weak automaton, the acceptance condition is $\alpha \subseteq Q$ and there exists a partition of $Q$ into disjoint sets, $Q_i$, such that for each set $Q_i$, either $Q_i \subseteq \alpha$, in which case $Q_i$ is an *accepting set*, or $Q_i \cap \alpha = \emptyset$, in which case $Q_i$ is a *rejecting set*. In addition, there exists a partial order $\leq$ on the collection of the $Q_i$'s such that for every $q \in Q_i$ and $q' \in Q_j$ for which $q'$ occurs in $\delta(q, \sigma, k)$, for some $\sigma \in \Sigma$ and $k \in \mathcal{D}$, we have $Q_j \leq Q_i$. Thus, transitions from a state in $Q_i$ lead to states in either the same $Q_i$ or a lower one. It follows that every infinite path of a run of a weak automaton ultimately gets "trapped" within some $Q_i$. The path then satisfies the acceptance condition if and only if $Q_i$ is an accepting set. Note that this corresponds to the Büchi acceptance condition. Indeed, a run visits infinitely many states in $\alpha$ iff it gets trapped in an accepting set.

The automaton $A_\psi$ defined in the proof of Theorem 6.1 is weak. To see this, consider the partition of $Q$ into disjoint sets in which each formula $\varphi \in cl(\psi)$ constitutes

a (singleton) set $\{\varphi\}$ in the partition. The partial order between the sets is then defined by $\{\varphi_1\} \leq \{\varphi_2\}$ iff $\varphi_1 \in cl(\varphi_2)$. Since each transition of the automaton from a state $\varphi$ leads to states associated with formulas in $cl(\varphi)$, the weakness conditions hold. In particular, each set is either contained in $\alpha$ or disjoint from $\alpha$.

As pointed out in [GO01], the fact $\mathcal{A}_\psi$ is weak (in fact, it is *very weak* – the sets $Q_i$ in the partition are singletons) enables a simpler removal of alternation than the one described in Theorem 6.2. In general, the construction we presented in Theorem 4.1 and the one that follows from the combination of Theorems 6.1 and 6.2 are very basic ones. Due to the heavy use of the construction in practice, numerous improvements have been suggested, cf. [GPVW95,SB00,GO01,Fri03].

## 7. Complementation of Büchi automata

In Section 3, we saw that NBW are closed under union and intersection. In this section we prove their closure under complementation.

The complementation problem for nondeterministic word automata has numerous applications in formal verification. In particular, the language-containment problem, to which many verification problems is reduced, involves complementation. For automata on finite words, which correspond to safety properties, complementation involves determinization. The $2^n$ blow-up that is caused by the subset construction is justified by a tight lower bound. For Büchi automata on infinite words, which are required for the modeling of liveness properties, optimal complementation constructions are quite complicated, as the subset construction is not sufficient. Efforts to develop simple complementation constructions for nondeterministic automata started early in the 60s, motivated by decision problems of second-order logics. Büchi suggested a complementation construction for nondeterministic Büchi automata that involved a complicated combinatorial argument and a doubly-exponential blow-up in the state space [Büc62]. Thus, complementing an automaton with $n$ states resulted in an automaton with $2^{2^{O(n)}}$ states. In [SVW87], Sistla et al. suggested an improved construction, with only $2^{O(n^2)}$ states, which is still, however, not optimal. Only in [Saf88], Safra introduced a determinization construction, which also enabled a $2^{O(n \log n)}$ complementation construction, matching a lower bound described by Michel [Mic88].

In this section we describe a complementation construction that avoids Safra's determinization. The construction, described in [KV01], uses instead intermediate universal co-Büchi automata and alternating weak automata. Here, we describe the construction without the intermediate automata, and go directly to a complementary NBW. The idea behind the construction is to assign ranks to nodes in a directed acyclic graph that embodies all the runs of the NBW. The idea can be applied also to richer types of acceptance conditions [KV05].

Let $\mathcal{A} = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$ be a nondeterministic Büchi automaton with $|Q| = n$, and let $w = \sigma_0 \cdot \sigma_1 \cdot$ be a word in $\Sigma^\omega$. We define an infinite DAG $G$ that embodies all the possible runs of $\mathcal{A}$ on $w$. Formally, $G = \langle V, E \rangle$, where

- $V \subseteq Q \times \mathbb{N}$ is the union $\bigcup_{l \geq 0}(Q_l \times \{l\})$, where $Q_0 = \{q_{in}\}$ and $Q_{l+1} = \bigcup_{q \in Q_l} \delta(q, \sigma_l)$.
- $E \subseteq \bigcup_{l \geq 0}(Q_l \times \{l\}) \times (Q_{l+1} \times \{l+1\})$ is such that $E(\langle q, l \rangle, \langle q', l+1 \rangle)$ iff $q' \in \delta(q, \sigma_l)$.

We refer to $G$ as the *run* DAG of $\mathcal{A}$ on $w$. We say that a vertex $\langle q', l' \rangle$ is a *successor* of a vertex $\langle q, l \rangle$ iff $E(\langle q, l \rangle, \langle q', l' \rangle)$. We say that $\langle q', l' \rangle$ is *reachable* from $\langle q, l \rangle$ iff there exists a sequence $\langle q_0, l_0 \rangle, \langle q_1, l_1 \rangle, \langle q_2, l_2 \rangle, \ldots$ of successive vertices such that $\langle q, l \rangle = \langle q_0, l_0 \rangle$, and there exists $i \geq 0$ such that $\langle q', l' \rangle = \langle q_i, l_i \rangle$. Finally, we say that a vertex $\langle q, l \rangle$ is an *$\alpha$-vertex* iff $q \in \alpha$. It is easy to see that $\mathcal{A}$ accepts $w$ iff $G$ has a path with infinitely many $\alpha$-vertices. Indeed, such a path corresponds to an accepting run of $\mathcal{A}$ on $w$.

A *ranking* for $G$ is a function $f : V \to [2n]$ that satisfies the following two conditions:

1. For all vertices $\langle q, l \rangle \in V$, if $f(\langle q, l \rangle)$ is odd, then $q \notin \alpha$.
2. For all edges $\langle \langle q, l \rangle, \langle q', l' \rangle \rangle \in E$, we have $f(\langle q', l' \rangle) \leq f(\langle q, l \rangle)$.

Thus, a ranking associates with each vertex in $G$ a rank in $[2n]$ so that the ranks along paths decreased monotonically, and $\alpha$-vertices get only even ranks. Note that each path in $G$ eventually gets trapped in some rank. We say that the ranking $f$ is an *odd ranking* if all the paths of $G$ eventually get trapped in an odd rank. Formally, $f$ is odd iff for all paths $\langle q_0, 0 \rangle, \langle q_1, 1 \rangle, \langle q_2, 2 \rangle, \ldots$ in $G$, there is $j \geq 0$ such that $f(\langle q_j, j \rangle)$ is odd, and for all $i \geq 1$, we have $f(\langle q_{j+i}, j+i \rangle) = f(\langle q_j, j \rangle)$.

**Lemma 7.1** *$\mathcal{A}$ rejects $w$ iff there is an odd ranking for $G$.*

**Proof:** We first claim that if there is an odd ranking for $G$, then $\mathcal{A}$ rejects $w$. To see this, recall that in an odd ranking, every path in $G$ eventually gets trapped in an odd rank. Hence, as $\alpha$-vertices get only even ranks, it follows that all the paths of $G$, and thus all the possible runs of $\mathcal{A}$ on $w$, visit $\alpha$ only finitely often.

Assume now that $\mathcal{A}$ rejects $w$. We describe an odd ranking for $G$. We say that a vertex $\langle q, l \rangle$ is *finite* in a (possibly finite) DAG $G' \subseteq G$ iff only finitely many vertices in $G'$ are reachable from $\langle q, l \rangle$. The vertex $\langle q, l \rangle$ is *$\alpha$-free* in $G'$ iff all the vertices in $G'$ that are reachable from $\langle q, l \rangle$ are not $\alpha$-vertices. Note that, in particular, an $\alpha$-free vertex is not an $\alpha$-vertex. We define an infinite sequence $G_0 \supseteq G_1 \supseteq G_2 \supseteq \ldots$ of DAGs inductively as follows.

- $G_0 = G$.
- $G_{2i+1} = G_{2i} \setminus \{\langle q, l \rangle \mid \langle q, l \rangle \text{ is finite in } G_{2i}\}$.
- $G_{2i+2} = G_{2i+1} \setminus \{\langle q, l \rangle \mid \langle q, l \rangle \text{ is } \alpha\text{-free in } G_{2i+1}\}$.

Consider the function $f : V \to \mathbb{N}$ where

$$f(\langle q, l \rangle) = \begin{bmatrix} 2i & \text{If } \langle q, l \rangle \text{ is finite in } G_{2i}. \\ 2i+1 & \text{If } \langle q, l \rangle \text{ is } \alpha\text{-free in } G_{2i+1}. \end{bmatrix}$$

Recall that $\mathcal{A}$ rejects $w$. Thus, each path in $G$ has only finitely many $\alpha$-vertices. It is shown in [KV01] that for every $i \geq 0$, the transition from $G_{2i+1}$ to $G_{2i+2}$ involves the removal of an infinite path from $G_{2i+1}$. Intuitively, it follows from the fact that as long as $G_{2i+1}$ is not empty, it contains a at least one $\alpha$-free vertex, from which an infinite path of $\alpha$-free vertices start. Since the width of $G_0$ is bounded by $n$, it follows that the width of $G_{2i}$ is at most $n - i$. Hence, $G_{2n}$ is finite, and $G_{2n+1}$ is empty. Thus, $f$ above maps the vertices in $V$ to $[2n]$. We claim further that $f$ is an odd ranking. First, since an $\alpha$-free vertex cannot be an $\alpha$-vertex and $f(\langle q, l \rangle)$ is odd only for $\alpha$-free vertices

$\langle q, l\rangle$, the first condition for $f$ being a ranking holds. Second, as argued in [KV01], for every two vertices $\langle q, l\rangle$ and $\langle q', l'\rangle$ in $G$, if $\langle q', l'\rangle$ is reachable from $\langle q, l\rangle$, then $f(\langle q', l'\rangle) \leq f(\langle q, l\rangle)$. In particular, this holds for $\langle q', l'\rangle$ that is a successor of $\langle q, l\rangle$. Hence, the second condition for ranking holds too. Finally, as argued in [KV01] for every infinite path in $G$, there exists a vertex $\langle q, l\rangle$ with an odd rank such that all the vertices $\langle q', l'\rangle$ in the path that are reachable from $\langle q, l\rangle$ have $f(\langle q', l'\rangle) = f(\langle q, l\rangle)$. Hence, $f$ is an odd ranking. □

By Lemma 7.1, an automaton $\mathcal{A}'$ that complements $\mathcal{A}$ can proceed on an input word $w$ by guessing an odd ranking for the run DAG of $\mathcal{A}$ on $w$. We now define such an automaton $\mathcal{A}'$ formally. We first need some definitions and notations.

A *level ranking* for $\mathcal{A}$ and $w$ is a function $g : Q \to [2n] \cup \{\bot\}$, such that if $g(q)$ is odd, then $q \notin \alpha$. Let $\mathcal{R}$ be the set of all level rankings. For two level rankings $g$ and $g'$, we say that $g'$ *covers* $g$ if for all $q$ and $q'$ in $Q$, if $g(q) \geq 0$ and $q' \in \delta(q, \sigma)$, then $0 \leq g'(q') \leq g(q)$.

We define $\mathcal{A}' = \langle \Sigma, \mathcal{R} \times 2^Q, q'_{in}, \delta', \mathcal{R} \times \{\emptyset\}\rangle$, where

- $q'_{in} = \langle g_{in}, \emptyset\rangle$, where $g_{in}(q_{in}) = 2n$ and $g_{in}(q) = \bot$ for all $q \neq q_{in}$. Thus, the odd ranking that $\mathcal{A}'$ guesses maps the root $\langle q_{in}, 0\rangle$ of the run DAG to $2n$.
- For a state $\langle g, P\rangle \in \mathcal{R} \times 2^Q$ and a letter $\sigma \in \Sigma$, we define $\delta'(\langle g, P\rangle, \sigma)$ as follows.
  ◦ If $P \neq \emptyset$, then

$$\delta'(\langle g, P\rangle, \sigma) = \{\langle g', P'\rangle : g' \text{ covers } g, \text{ and } P' = \{q' : \text{ there is } q \in P \text{ such that } q' \in \delta(q, \sigma) \text{ and } g'(q') \text{ is even}\}\}.$$

  ◦ If $P = \emptyset$, then

$$\delta'(\langle g, P\rangle, \sigma) = \{\langle g', P'\rangle : g' \text{ covers } g, \text{ and } P' = \{q' : g'(q') \text{ is even}\}\}.$$

  Thus, when $\mathcal{A}'$ reads the $l$'th letter in the input, for $l \geq 1$, it guesses the level ranking for level $l$ in the run DAG. This level ranking should cover the level ranking of level $l-1$. In addition, in the $P$ component, $\mathcal{A}'$ keeps track of states whose corresponding vertices in the DAG have even ranks. Paths that traverse such vertices should eventually reach a vertex with an odd rank. When all the paths of the DAG have visited a vertex with an odd rank, the set $P$ becomes empty, and is initiated by new obligations for visits in odd ranks according to the current level ranking. The acceptance condition $\mathcal{R} \times \{\emptyset\}$ then checks that there are infinitely many levels in which all the obligations have been fulfilled.

## 8. Exercises

*Question 1*

For each pair $\varphi_1; \varphi_2$ of formulas below, decide which of the following hold (note that possibly both $a$ and $b$ hold, or none of them):

**a.** $\varphi_1 \to \varphi_2$.

**b.** $\varphi_1 \leftarrow \varphi_2$.

When the an implication does not hold, describe a counter example (when $a$ does not hold, describe a model for $\varphi_1$ that does not satisfy $\varphi_2$, and when $b$ does not hold, describe a model for $\varphi_2$ that does not satisfy $\varphi_1$).

$$\begin{array}{ll} \mathbf{1}.\, Gp & ; \neg FX\neg p \\ \mathbf{2}.\, G(p \vee q) & ; Gp \vee Gq \\ \mathbf{3}.\, G(p \wedge q) & ; Gp \wedge Gq \\ \mathbf{4}.\, qUp & ; q \wedge XqUp \\ \mathbf{5}.\, pU(qUr) & ; (pUq)Ur \\ \mathbf{6}.\, p \wedge Xq & ; pUq \wedge qUp \end{array}$$

*Question 2*

Describe nondeterministic Büchi automata for the following properties.

1. $F(p \wedge Xp)$.
2. $FG(p \vee Xp)$.
3. $Gp \rightarrow Gq$.
4. $GFp \rightarrow GFq$.

*Question 3*

Prove or give a counter example:

1. Every nondeterministic Büchi automaton $\mathcal{A}$ has an equivalent nondeterministic Büchi automaton $\mathcal{A}'$ with a single initial state.
2. Every nondeterministic Büchi automaton $\mathcal{A}$ has an equivalent nondeterministic Büchi automaton $\mathcal{A}'$ with a single accepting state.

*Question 4*

Given a generalized Büchi automaton with $n$ states and index $k$, construct an equivalent Büchi automaton with $nk$ states.

*Question 5*

Given a deterministic Büchi automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \alpha \rangle$ with $n$ states, describe a nondeterministic Büchi automaton $\mathcal{A}'$ with $O(n)$ states such that $\mathcal{L}(\mathcal{A}') = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.

Hint: the NBW $\mathcal{A}'$ uses its nondeterminism in order to guess when the run of $\mathcal{A}$ stops visiting $\alpha$.

*Question 6*

Consider a nondeterministic word automaton $\mathcal{A}$. Let $\mathcal{L}_*(\mathcal{A})$ be the language of $\mathcal{A}$ when regarded as an automaton on finite words, and let $\mathcal{L}_\omega(\mathcal{A})$ be the language of $\mathcal{A}$ when regarded as a Büchi automaton. Prove or give a counter example:

1. $\mathcal{L}_\omega(\mathcal{A}) = lim(\mathcal{L}_*(\mathcal{A}))$.
2. $\mathcal{L}_\omega(\mathcal{A}) = lim(\mathcal{L}_*(\mathcal{A}))$ iff there is some deterministic Büchi automaton that recognizes $\mathcal{L}_\omega(\mathcal{A})$.

*Question 7*

In a *co-Büchi* word automaton, the acceptance condition is a set $\alpha \subseteq Q$ and a run $r$ is accepting iff it visits $\alpha$ only finitely often; that is $inf(r) \cap \alpha = \emptyset$.

1. Prove that a language $L$ is recognizable by a deterministic Büchi automaton iff $\Sigma^\omega \setminus L$ is recognizable by a deterministic co-Büchi automaton.
2. Given a nondeterministic co-Büchi automaton $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, define a deterministic co-Büchi automaton $\mathcal{A}' = \langle \Sigma, Q', \delta', q_0', \alpha' \rangle$ equivalent to $\mathcal{A}$.
Hint: $Q' = 2^Q \times 2^Q$, and all the reachable states $\langle S, P \rangle$ in $Q'$ are such that $P \subseteq S$. The acceptance condition $\alpha' = 2^Q \times \{\emptyset\}$.
3. Let $L = \{w : w$ has infinitely many $a$'s $\} \subseteq \{a, b\}^\omega$. Can $L$ be recognized by a nondeterministic co-Büchi word automaton? Justify your answer.

*Question 8*

In this question we prove an exponential lower bound on the translation of LTL to nondeterministic Büchi automata. Let $AP = \{p, q\}$. For $n \geq 1$, we define the language $\mathcal{L}_n$ over the alphabet $2^{AP}$ as follows.

$$\mathcal{L}_n = \{\{\{p\},\{q\},\emptyset,\}^* \cdot \emptyset \cdot w \cdot \emptyset \cdot \{\{p\},\{q\},\emptyset,\}^* \cdot \{p,q\} \cdot w \cdot \{p,q\}^\omega : w \in \{\{p\},\{q\}\}^n\}.$$

Thus, a word is in $\mathcal{L}_n$ iff the word between the first and second $\{p, q\}$ is of length $n$, it is composed of letters in $\{\{p\}, \{q\}\}$ only, and it has appeared between $\emptyset$'s somewhere before the first $\{p, q\}$. In addition, the second $\{p, q\}$ starts an infinite tail of $\{p, q\}$'s.

1. Describe two words in $\mathcal{L}_3$ and two words not in $\mathcal{L}_3$.
2. Prove that the smallest nondeterministic Büchi automaton that recognizes $\mathcal{L}_n$ has at least $2^n$ states.
3. Specify $\mathcal{L}_n$ with an LTL formula of length quadratic in $n$.

*Question 9*

Consider the translation of an LTL formula $\psi$ to nondeterministic Büchi automata $\mathcal{A}_\psi$ we saw in class. Recall that $\mathcal{A}_\psi^S$ (that is, $\mathcal{A}_\psi$ with initial state $S \subseteq cl(\psi)$) accepts exactly these words in $(2^{AP})^\omega$ that satisfy exactly all the formulas in $S$.

1. Construct the automaton for $\psi = F(p \wedge Xp)$. Note you are asked to construct exactly the automaton we saw in class (which may not be the minimal automaton for $\psi$).
2. Describe a linear-time procedure for complementing the automaton $\mathcal{A}_\psi$ (for an arbitrary $\psi$).
3. Which of the following statements are correct? (prove or give a counter example)

   (a) For every LTL formula $\psi$, if there is a deterministic Büchi automaton for $\psi$, then $\mathcal{A}_\psi$ is deterministic.

(b) For every LTL formula $\psi$ and a word $w \in (2^{AP})^\omega$ such that $w \models \psi$, there is a single run of $\mathcal{A}_\psi$ that accepts $w$.

*Question 10*

Describe an ABW with $O(n)$ states for the language $\mathcal{L}_n = \{w \cdot w \cdot \#^\omega : w \in (0+1)^n\}$.

*Question 11*

For a word $w \in \Sigma^\omega$, let $\mathit{suff}(w) = \{y : x \cdot y = w$, for some $x \in \Sigma^*\}$. Note that $\mathit{suff}(w)$ contains $w$ and $\epsilon$. For a language $L \subseteq \Sigma^\omega$, let $\mathit{suff_limit}(L) = \{w : \mathit{suff}(w) \subseteq L\}$. Thus, $\mathit{suff_limit}(L)$ contains exactly all words $w$ such that all the suffixes of $w$ are in $L$.

Given an NBW $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, describe an ABW $\mathcal{A}'$ with the same state space $Q$ such that $\mathcal{L}(\mathcal{A}') = \mathit{suff_limit}(\mathcal{L}(\mathcal{A}))$. Prove the correctness of the construction formally.

## References


[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. International Congress on Logic, Method, and Philosophy of Science. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[Cho74] Y. Choueka. Theories of automata on $\omega$-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981.

[Fri03] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating b§chi automata. In *Proc. 8th Intl. Conference on Implementation and Application of Automata*, number 2759 in Lecture Notes in Computer Science, pages 35–48. Springer-Verlag, 2003.

[FW02] C. Fritz and T. Wilke. State space reductions for alternating Büchi automata: Quotienting by simulation equivalences. In *Proc. 22th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *Lecture Notes in Computer Science*, pages 157–169, December 2002.

[GKSV03] S. Gurumurthy, O. Kupferman, F. Somenzi, and M.Y. Vardi. On complementing nondeterministic Büchi automata. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 96–110. Springer-Verlag, 2003.

[GO01] P. Gastin and D. Oddoux. Fast LTL to büchi automata translation. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer-Verlag, 2001.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, August 1995.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[KV01] O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. on Computational Logic*, 2(2):408–429, July 2001.

[KV05] O. Kupferman and M.Y. Vardi. Complementation constructions for nondeterministic automata on infinite words. In *Proc. 11th International Conf. on Tools and Algorithms for The Construction*

*and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 206–221. Springer-Verlag, 2005.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Lan69] L.H. Landweber. Decision problems for $\omega$–automata. *Mathematical Systems Theory*, 3:376–384, 1969.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[McN66] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on $\omega$-words. *Theoretical Computer Science*, 32:321–330, 1984.

[Mic88] M. Michel. Complementation is more difficult with automata on infinite words. CNET, Paris, 1988.

[MSS86] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *Proc. 13th International Colloquium on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 275 – 283. Springer-Verlag, 1986.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[Saf88] S. Safra. On the complexity of $\omega$-automata. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 319–327, White Plains, October 1988.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Computer Aided Verification, Proc. 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer-Verlag, 2000.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Practical Principles for Computer Security

Butler LAMPSON[1]
*Microsoft Research*
*Marktoberdorf, 2006*

What do we want from secure computer systems? Here is a reasonable goal:

*Computers are as secure as real world systems, and people believe it.*

Most real world systems are not very secure by the absolute standard suggested above. It's easy to break into someone's house. In fact, in many places people don't even bother to lock their houses, although in Manhattan they may use two or three locks on the front door. It's fairly easy to steal something from a store. You need very little technology to forge a credit card, and it's quite safe to use a forged card at least a few times.

**Real security is about punishment, not about locks;**
**about accountability, not access control**

Why do people live with such poor security in real world systems? The reason is that real world security is not about perfect defenses against determined attackers. Instead, it's about

- value,
- locks, and
- punishment.

The bad guys balance the value of what they gain against the risk of punishment, which is the cost of punishment times the probability of getting punished. The main thing that makes real world systems sufficiently secure is that bad guys who do break in are caught and punished often enough to make a life of crime unattractive. The purpose of locks is not to provide absolute security, but to prevent casual intrusion by raising the threshold for a break-in.

**Security is about risk management**

Well, what's wrong with perfect defenses? The answer is simple: they cost too much. There is a good way to protect personal belongings against determined attackers: put them in a safe deposit box. After 100 years of experience, banks have learned how to use steel and concrete, time locks, alarms, and multiple keys to make these boxes quite secure. But they are both expensive and inconvenient. As a result, people use them only for things that are seldom needed and either expensive or hard to replace.

Practical security balances the cost of protection and the risk of loss, which is the cost of recovering from a loss times its probability. Usually the probability is fairly small (because the risk of punishment is high enough), and therefore the risk of loss is also small. When the risk is less than the cost of recovering, it's better to accept it as a cost of doing business (or a cost of daily living) than to pay for better security. People and credit card companies make these decisions every day.

[1]My colleagues Martin Abadi, Carl Ellison, Charlie Kaufman, and Paul Leach made many suggestions for improvement and clarification. Some of these ideas originated in the Taos authentication system ([4], [6])

With computers, on the other hand, security is only a matter of software, which is cheap to manufacture, never wears out, and can't be attacked with drills or explosives. This makes it easy to drift into thinking that computer security can be perfect, or nearly so. The fact that work on computer security has been dominated by the needs of national security has made this problem worse. In this context the stakes are much higher and there are no police or courts available to punish attackers, so it's more important not to make mistakes. Furthermore, computer security has been regarded as an offshoot of communication security, which is based on cryptography. Since cryptography can be nearly perfect, it's natural to think that computer security can be as well.

What's wrong with this reasoning? It ignores two critical facts:

- Secure systems are complicated, hence imperfect.
- Security gets in the way of other things you want.

The end result should not be surprising. We don't have "real" security that guarantees to stop bad things from happening, and the main reason is that people don't buy it. They don't buy it because the danger is small, and because security is a pain.

- Since the danger is small, people prefer to buy features. A secure system has fewer features because it has to be implemented correctly. This means that it takes more time to build, so naturally it lacks the latest features.
- Security is a pain because it stops you from doing things, and you have to do work to authenticate yourself and to set it up.

A secondary reason we don't have "real" security is that systems are complicated, and therefore both the code and the setup have bugs that an attacker can exploit. This is the reason that gets all the attention, but it is not the heart of the problem.

## 1. Implementing security

The job of computer security is to defend against vulnerabilities. These take three main forms:
1) Bad (buggy or hostile) *programs*.
2) Bad (careless or hostile) agents, either programs or *people*, giving bad instructions to good but gullible programs.
3) Bad agents tapping or spoofing *communications*.
Case (2) can be cascaded through several levels of gullible agents. Clearly agents that might get instructions from bad agents must be prudent, or even paranoid, rather than gullible.
Broadly speaking, there are five defensive strategies:
4) *Coarse: Isolate*—keep everybody out. It provides the best security, but it keeps you from using information or services from others, and from providing them to others. This is impractical for all but a few applications.
5) *Medium: Exclude*—keep the bad guys out. It's all right for programs inside this defense to be gullible. Code signing and firewalls do this.
6) *Fine: Restrict*—Let the bad guys in, but keep them from doing damage. Sandboxing does this, whether the traditional kind provided by an operating system process, or the modern kind in a Java virtual machine. Sandboxing typically involves access control on

resources to define the holes in the sandbox. Programs accessible from the sandbox must be paranoid; it's hard to get this right.

7) *Recover*—Undo the damage. Backup systems and restore points are examples. This doesn't help with secrecy, but it helps a lot with integrity and availability.

8) *Punish*—Catch the bad guys and prosecute them. Auditing and police do this.

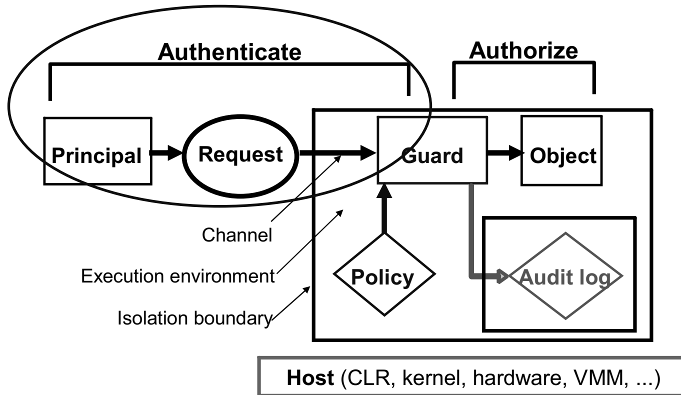



**Figure 1.** Access control model

The well-known *access control* model shown in Figure 1 provides the framework for these strategies. In this model, a guard controls the access of requests for service to valued resources, which are usually encapsulated in objects. The guard's job is to decide whether the source of the request, called a *principal*, is allowed to do the operation on the object. To decide, it uses two kinds of information: *authentication* information from the left, which identifies the principal who made the request, and *authorization* information from the right, which says who is allowed to do what to the object. There are many ways to make this division. The reason for separating the guard from the object is to keep it simple.

Of course security still depends on the object to implement its methods correctly. For instance, if a file's read method changes its data, or the write method fails to debit the quota, or either one touches data in other files, the system is insecure in spite of the guard.

Another model is sometimes used when secrecy in the face of bad programs is a primary concern: the *information flow control* model shown in Figure 2 [5]. This is roughly a dual of the access control model, in which the guard decides whether information can flow to a principal.

In either model, there are three basic mechanisms for implementing security. Together, they form the gold standard for security (since they all begin with Au):

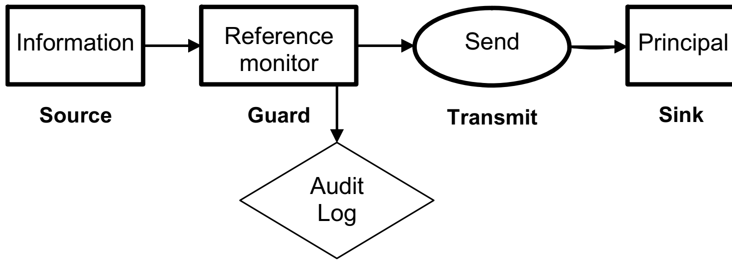



**Figure 2.** Information flow model

- ***Au****thenticating* principals, answering the question "Who said that?" or "Who is getting that information?". Usually principals are people, but they may also be groups, machines, or programs.
- ***Au****thorizing* access, answering the question "Who is trusted to do which operations on this object?".
- ***Au****diting* the decisions of the guard, so that later it's possible to figure out what happened and why.

## 2. Access control

Figure 1 shows the overall model for access control. It says that *principals* make *requests* on *objects*; this is the basic paradigm of object-oriented programming or of services. The job of security is to decide whether a particular request is allowed; this is done by the *guard*, which needs to know who is making the request (the principal), what the request is, and what the target of the request is (the object). The guard is often called the *relying party*, since it relies on the information in the request and in policy to make its decision. Because all trust is local, the guard has the final say about how to interpret all the incoming information. For the guard to do its job it needs to see every request on the object; to ensure this the object is protected by an *isolation boundary* that blocks all access to the object except over a channel that passes through the guard. There are many ways to implement principals, requests, objects and isolation, but this abstraction works for all of them.

The model has three primary elements:

1. **Isolation**: This constrains the attacker to enter the protected execution environment via access-controlled channels.
2. **Access Control**: Access control is broken down into authentication, authorization, and auditing.
3. **Policy and User Model**: Access control policy is set by human beings—sometimes trained, sometimes not.

This paper addresses one piece of the security model: access control. It gives an overview that extends from setting authentication policy through authenticating a request

to the mechanics of checking access. It then discusses the major elements of authentication and authorization in turn.

### *2.1. What is access control*

Every action that requires a security decision, whether it is a user command, a system call, or the processing of a message from the net, is represented in the model of as a request from a principal over a channel. Each request must pass through a guard or relying party that makes an access control decision. That decision consists of a series of steps:

1. Do *direct* authentication, which establishes the principal directly making the request. The most common example of this is verifying a cryptographic signature on a message; in this case the principal is the cryptographic key that verifies the signature. Another example is accepting input from the keyboard, which is the principal directly making the request.[2]
2. (optionally) Associate one or more other principals with the principal of step 1. These could be groups or attributes.
3. Do authorization, which determines whether any of these principals is allowed to have the request fulfilled on that object.

The boundary between authentication and authorization, however, is not clear. Different experts draw it in different places. It is also not particularly relevant, since it makes little sense to do one without the other.

## 3. Examples: Logon and cross-organization access control

This section gives two examples to introduce the basic ideas of access control.

### *3.1. Example: User and network logon*

Figure 3 shows the basic elements of authentication and how they are used to log on a user, access a resource, and then do a network logon to another host. Note the distinction between the elements that are part of a single host and external token sources such as domain controllers and STS's. For concreteness, the figure describes the process of authenticating a user as logon to Windows, that is, as creating a Windows session that can speak for the user; in Windows a SID is a 128-bit binary identifier for a principal. However, exactly the same mechanisms can be used to log onto an application such as SQL Server, or to authenticate a single message, so it covers these cases equally well.

The numbers in the figure label the steps of the logon, which are as follows:

1. The user provides some input for logon (for example, user name and password).
2. The logon agent sends a logon validation request with the input (or something derived from it) to the domain controller (labeled "token source" in the figure),
3. which replies with the user's SID and a session key if logon succeeded, and an error if it didn't.

[2]See the appendix for a sketch of what you need to know about cryptography.

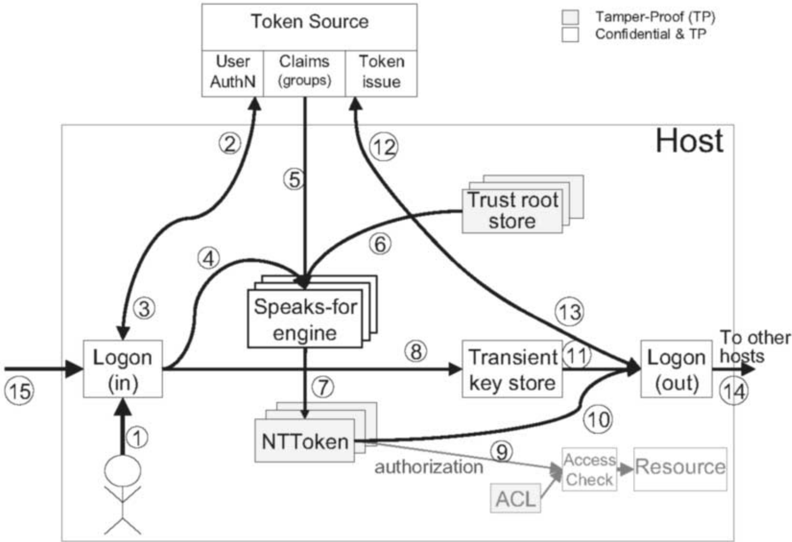



**Figure 3.** Core logon example

4. The token source provides the user's SID,
5. and uses it to provide the group SIDs.
6. The trust root says that the token source should be trusted to logon anyone, so
7. all the SIDs go into the NT token,
8. and the session key is saved in the transient key store.
9. When the process accesses some local resource the NT token is checked against the ACL, and with luck the access is granted.
10. When the process wants to access a remote resource, the NT token
11. and the session key are needed
12. to ask the token source to
13. issue a token that can be sent out
14. to the remote host,
15. which receives it (back on the left side of the figure) and does a net logon.

### *3.2. Example: Cross-organization access control*

A distributed system may involve systems (and people) that belong to different organizations and are managed differently. To do access control cleanly in such a system (as opposed to the local systems that are well supported by Windows domains, as in the previous example) we need a way to treat uniformly all the information that contributes to the decision to grant or deny access. Consider the following example, illustrated in Figure 4.

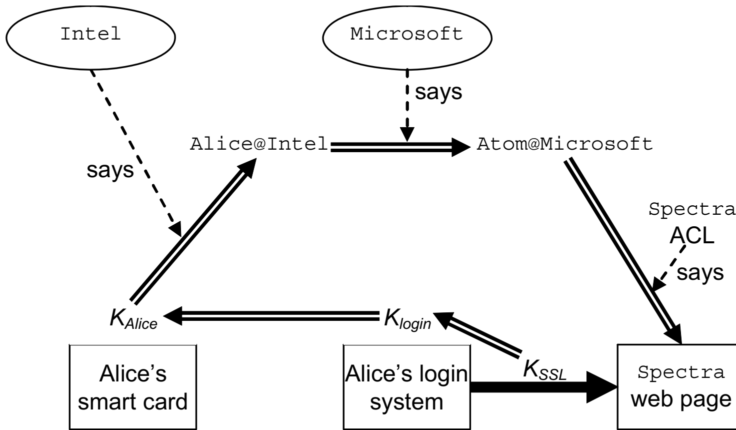



**Figure 4.** Speaks-for example

Alice at Intel is part of a team working on a joint Intel-Microsoft project called Atom. She logs in to her Intel workstation, using a smart card to authenticate herself, and connects using SSL to a project web page called Spectra at Microsoft. The web page grants her access because:

1. The request comes over an SSL connection secured with a connection key $K_{SSL}$ created using the Diffie-Hellman key exchange protocol.
2. To authenticate the SSL connection, Alice's workstation uses its temporary logon key $K_{logon}$ to sign a statement certifying that requests secured by the connection key $K_{SSL}$ come from the logon session.[3]
3. At logon time, Alice's smart card uses her key $K_{Alice}$ certifies that requests signed by the logon session key $K_{logon}$ come from Alice.
4. Intel certifies that $K_{Alice}$ is the key for Alice@Intel.com.[4]
5. Microsoft's group database says that Alice@Intel.com is in the Atom group.
6. The ACL on the Spectra page says that Atom has read/write access.

In the figure, Alice's requests to Spectra travel over the SSL channel (represented by the fat arrow), which is secured by the key $K_{SSL}$. In contrast, the reasoning about trust that allows Spectra to conclude that it should grant the requests runs clockwise around the circle of double arrows; note that requests never travel on this path.

From this example we can see that many different kinds of information contribute to the access control decision:

- Authenticated session keys

[3]Saying that the workstation signs with the public key $K_{logon}$ means that it encrypts with the corresponding private key. Through the magic of public-key cryptography, anyone who knows the public key can verify this signature. This is not the only way to authenticate an SSL connection, but it is the simplest to explain.

[4]Intel can do this with an X.509 certificate, or by responding to a query "Is $K_{Alice}$ the key for Alice@Intel.com?", or in some other secure way.

- User passwords or public keys
- Delegations from one system to another
- Group memberships
- ACL entries.

We want to do a number of things with this information:

- Keep track of how secure channels are authenticated, whether by passwords, smart cards, or systems.
- Make it secure for Microsoft to accept Intel's authentication of Alice.
- Handle delegation of authority to a system, for example, Alice's logon system.
- Handle authorization via ACLs like the one on the Spectra page.
- Record the reasons for an access control decision so that it can be audited later.

## 4. Basic concepts

This section describes the basic concepts, informally but in considerable detail: principals and identifiers; speaks-for and trust; tokens; paths, security domains, attributes, and groups; global identifiers; how to choose identifiers and names, and freshness or consistency. Sections 5 and 6 describe the components of the architecture and how they use these concepts.

### *4.1. Principals and identifiers*

A principal is the source of a request in the model of ; it is the answer to the questions:

- "Who made this request?" (authentication)
- "Who is trusted for this request?" (authorization-for example, who is on the ACL)

We say that the principal says the request, as in *P* **says do** read report.doc. In addition to saying requests, principals can also say speaks-for statements or claims, as explained in section 4.2.

Principals are not only people and devices. Executable code is a principal. An input/output channel and a cryptographic signing key are principals. So are groups such as Microsoft-FTE and attributes such as age=32. We treat all these uniformly because they can all be answers to the question "Who is trusted for this request?". Furthermore, if we interpret the question "Who made this request?" broadly, they can all be answers to this question as well: a request can be made directly only by a channel or key, but it can be made indirectly by a person (or device) that controls the key, or by a group that such a person is a member of.

It turns out to be convenient to treat objects or resources as principals too, even though they don't make requests.

Principals can be either simple or compound. Simple principals are denoted by identifiers, which are strings. Intuitively, *identifiers* are labels used for people, computers and other devices, applications, attributes, channels, resources, etc., or groups of these.[5] Compound principals are explained in section 5.8.

[5]Programs usually can deal only with identifiers, not with the real-world principals that they denote. In this paper we will ignore this distinction for the most part.

Channels are special because they are the only *direct* principals: a computer can tell directly that a request comes from a channel, without any other information. Thus any authentication of a request must start with a channel. A cryptographic signing key is the most important kind of channel.

An identifier is a string; often the string encodes a path, as explained below. The string can be meaningful (to humans), or it can be meaningless; for example, it can encode a binary number (Occasionally an identifier is something that is meaningful, but not as a string of characters, such as a picture.). This distinction is important because access control policy must be expressed in terms of meaningful identifiers so that people can understand it, and also because people care about the meanings of a meaningful identifier such as coke.com, but no one cares about the bit pattern of a binary identifier. Of course there are gray areas in this taxonomy; a name such as davcdata.exe is not meaningful to most people, and a phone number might be very meaningful. But the taxonomy is useful none the less.

Meaningless identifiers in turn can be direct or not. This leads to a three-way classification of identifiers:

- *name*: an identifier that is meaningful to humans.
- *ID*: a meaningless identifier that is not direct. In this taxonomy an identifier such as xpz5914@hotmail.com is probably an ID, not a name, since it probably isn't meaningful.
- *direct*: a meaningless identifier that identifies a channel. There are three kinds of direct identifiers:
  - *key*: a cryptographic key (most simply, a public key) that can verify a signature on a request. We view a signing key as a channel, and say that messages signed by the key arrive on the channel named by that key.[6]
  - *hash*: a cryptographic collision-free hash of data (code, other files, keys, etc.): different data is guaranteed to have different hashes. A hash $H$ can say X if a suitable encoding of "This data **says** X" appears in the data of which $H$ is the hash. For code we usually hash a *manifest* that includes the hash of each member file. This has the same collision-free property as a hash of the contents of all the files.
  - *handle*: an identifier provided by the host for some channel, such as the keyboard (Strictly speaking, the wire from the keyboard.) or a pipe.

An identifier can be a *path*, which is a sequence of strings, just like a path name for a file such as C:\program files\Adobe\Acrobat6. It can be encoded as a single string using some syntactic convention. There are a number of different syntactic conventions for representing a path as a single string; the file name example uses "\" as a separator. The canonical form is left-to-right with / as the separator. A path can be rooted in a key, such as $K_{Verisign}$/andy@intel.com (or $K_{Verisign}$/com/intel/andy in the canonical form for paths); such a path is called *fully qualified*. A path not rooted in a key is rooted in self, the local environment interpreting the identifier; it is like a relative file name because its meaning depends on the context.

[6]For a symmetric key we can use a hash of it as the public name of the channel, though of course this is not enough to verify a signature.

## 4.2. Speaks-for and trust

Authentication must start with a channel, for example, with a cryptographic signature key. But it must end up with access control policy, which has to be expressed in terms of names so that people can understand it. To bridge the gap between channels and names we uses the notion of "speaks-for". We say that a channel speaks for a user, for example, if we trust that every request that arrives on the channel comes from the user, in other words, if the channel is trusted to *speak for* the user.

But the notion of speaks-for is much more general than this, as the example of section 3 illustrates. What is the common element in all the steps of the example and all the different kinds of information? There is a *chain of trust* running from the request at one end to the Spectra resource at the other. A link of this chain has the form

"Principal *P* speaks for principal *Q* about statements in set *R*"

For example, $K_{SSL}$ speaks for $K_{Alice}$ about everything, and Atom@Microsoft speaks for Spectra about read and write. We write "about *R*" as shorthand for "about statements in set *R*". Often *P* is called the *subject* and *R* is called the *rights*.

The idea of "*P* speaks for *Q* about *R*" is that

**if *P* says something about *R*, then *Q* says it too**

That is, *P* is trusted as much as *Q*, at least for statements in *R*. Put another way, *Q* takes responsibility for anything that *P* says about *R*. A third way: *P* is a more powerful principal than *Q* (at least with respect to *R*) since *P*'s statements are taken at least as seriously as *Q*'s (and perhaps more seriously). Thus *P* has all of *Q*'s authority about *R*.

The notion of principal is very general, encompassing any entity that we can imagine making statements or being trusted. Secure channels, people, groups, attributes, systems, program images, and resource objects are all principals. The notion of speaks-for is also very general; some examples are:

Binding a key to a user name.

Binding a program hash to a name for the program.

Allowing an authority to certify a set of names.

Making a user a member of a group.

Assigning a principal an attribute.

Granting a principal access to a resource by putting it on the resource's ACL.

The idea of "about *R*" is that *R* is some way of describing a set of things that *P* (and therefore *Q*) might say. You can think of *R* as a pattern or predicate that characterizes this set of statements, or you can think of it as some rights that *P* can exercise as much as *Q* can. In the example of section 3, *R* is "all statements" except for step (5), where it is "read and write requests". It's up to the guard of the object that gets the request to figure out whether the request is in *R*, so the interpretation of *R*'s encoding can be local to the object. For example, we could refine "read and write requests" to "read and write requests for files whose names match /users/lampson/security/*.doc". In most ACEs today, *R* is encoded as a bit vector of permissions, and you can't say anything as complicated as the previous sentence.

We can write this $P \Rightarrow_R Q$ for short, or just $P \Rightarrow Q$ without any subscript if *R* is "all statements". With this notation the chain for the example is:

$K_{SSL} \Rightarrow K_{logon} \Rightarrow K_{Alice} \Rightarrow$ Alice@Intel $\Rightarrow$ Atom@Microsoft $\Rightarrow_{r/w}$ Spectra

A single speaks-for fact such as $K_{Alice} \Rightarrow$ Alice@Intel is called a *claim*. The principal on the left is the *subject*.

The way to think about it is that ⇒ is "greater than or equal": the more powerful principal goes on the left, and the less powerful one on the right. So role=architect ⇒ Slava means that everyone in the architect role has all the power that Slava has. This is unlikely to be what you want. The other way, Slava ⇒ role=architect, means that Slava has all the power that the architect role has. This is a reasonable way to state the implications for security of making Slava an architect.

Figure 4 shows how the chain of trust is related to the various principals. Note that the "speaks for" arrows are quite independent of the flow of bytes: trust flows clockwise around the loop, but no data traverses this path. The example shows that claims can abstract from a wide variety of real-world facts:

- A key can speak for a person ($K_{Alice} \Rightarrow$ Alice@Intel) or for a naming authority ($K_{Intel} \Rightarrow$ Intel.com).
- A person can speak for a group (Alice@Intel ⇒ Atom@Microsoft).
- A person or group can speak for a resource, usually by being on the ACL of the resource (Atom@Microsoft $\Rightarrow_{r/w}$ Spectra). We say that Spectra makes this claim by putting Atom on its ACL.

### *4.2.1. Establishing claims: Delegation*

How does a claim get established? It can be built in; such facts appear in the trust root, discussed in section 5.1. Or it can be derived from other claims, or from statements made by principals, according to a few simple rules:

(S1) Speaks-for is transitive: if $P \Rightarrow Q$ and $Q \Rightarrow R$ then $P \Rightarrow R$.

(S2) A principal speaks for any path rooted in itself: $P \Rightarrow P/N$. This is just like a file system, where a directory controls its contents. Section 4.1 discusses paths.

(S3) Principals are trusted to *delegate* their authority, privileges, rights, etc.: if $Q$ **says** $P \Rightarrow Q$ then $P \Rightarrow Q$. (There are restricted forms of speaks-for where this rule doesn't hold.)

From the definition of ⇒, if $Q'$ **says** $P \Rightarrow Q$ and $Q' \Rightarrow Q$ then $Q$ **says** $P \Rightarrow Q$, and it follows from (S3) that $P \Rightarrow Q$. So a principal is trusted to delegate the authority of any principal it speaks for, not just its own authority. Frequently a delegation is restricted so that the delegate $P$ speaks for $Q$ only for requests (this is the usual interpretation of an X.509 end-entity certificate, for example, or membership in a group) or only for further delegation (an X.509 CA certificate, or GROUP_ADD/REMOVE_MEMBER permission on the ACL for a group).

### *4.2.2. Validity period*

A claim usually has a validity period, which is an interval of real time during which it is valid. When applying the rules to derive a claim from other claims and tokens, intersect their validity periods to get the validity period of the derived claim. This ensures that the derived claim is only valid when all of the inputs to its derivation are valid. A claim can be the result of a query to some authority A. For example, if the result of a query "Is $P$ in group $G$" to a database of group memberships is "Yes", that is an encoding of the claim $P \Rightarrow G$. The validity period of such a statement is often just the instant at which the response is made, although the queryer might choose to cache it and believe it for a longer time.

### 4.3. Tokens

A claim made by a principal is called a *token* (not to be confused with a user authentication token such as a SecurID device). Many tokens are called certificates, but this paper uses the more general term except when discussing X.509 certificates specifically. The rule (S3) tells you whether or not to believe a token; section 4.5 on global identifiers gives the most important example of this.

Examples of tokens:

1. X.509 certificate [ $K_I$ **says** $K_S \Rightarrow$ name, (optionally $K_I$ **says** name $\Rightarrow$ attribute)]
2. Authenticode certificate [ $K_V$ **says** H(code) $\Rightarrow$ publisher/program ]
3. Group memberships [ $K_D$ **says** $S_U \Rightarrow S_G$ ]
4. Signed SAML attribute assertion [ $K_I$ **says** name $\Rightarrow$ attribute ]
5. ISO REL (XrML) license

where $K_I$ is the issuer key, $K_S$ is the subject key, "name" is the certified name, $K_V$ is Verisign's key, H(code) is the hash value of the code being signed, "publisher" is the name of the code's publisher, $K_D$ is the key of the domain controller, $S_U$ is the SID of the user and $S_G$ is the SID of the group of which the user is a member. XrML tokens can do all of these things, and more besides.

A token can be signed in several different ways, which don't change the meaning of a token to its intended recipient, but do affect how difficult it is to forward:

- A token signed by a public key, like a X.509 certificate, can be forwarded to anyone without the cooperation of the third party. From a security point of view it is like a broadcast.
- A token signed by a symmetric key, like a Kerberos ticket, can be returned to its sender for forwarding to anyone with whom the sender shares a symmetric key.
- A token that is just sent on an authenticated channel cannot be forwarded, since there's no way to prove to anyone that the sender said it.

In a token the principals on both sides of the $\Rightarrow$ must be represented by identifiers, and it's important for these identifiers to be unambiguous. A fully qualified identifier (one that starts with a key or hash) is unambiguous. Other identifiers depend on the context, that is, on some convention between the issuer and the consumer of the token.

Like a claim, a token usually has a validity period; see section 4.2.2. For example, a Kerberos token is typically valid for eight hours.

A token is the most common way for a principal to communicate a claim to others, but it is not the only way. You can ask a principal *A* "Do you say $P \Rightarrow Q$?" or "What principal does *P* speak for?" and get back "*A* **says** 'yes'" or "*A* **says** '*Q*'". Such a statement only makes sense as a response to the original query; to be secure it must not only be signed by (some principal that speaks for) *A*, but also be bound securely to the query (for example, by a secure RPC protocol), so that an adversary can't later supply it as the re-sponse to some other query.

### 4.4. Organizing principals

There are several common ways to impose structure on principals in addition to the path identifiers introduced in section 4.1: security domains, attributes, and groups.

#### *4.4.1. Security domains*

A security domain is a collection of principals (users, groups, computers, servers and other resources) to which a particular set of policies apply, or in other words, that have common management. Usually we will just say "domain". It normally comprises:

- A key $K_D$.
- A namespace based on that key.
- A trust root—a set of claims of the form $K_{j1} \wedge K_{j2} \ldots \Rightarrow$ identifier-pattern
- ACLs for the trust root and the accounts, which define the administrators of the domain.
- A set of *accounts*—statements of the form $K_D$ says $K_i \Rightarrow K_D/N$ for principals with names in its namespace.
- A set of resources and policies for those resources

The essential property of paths is that namespaces with different roots are independent, just as different file system volumes are independent. In fact, namespaces with different *prefixes* are independent, just as file system directories with different names are independent. This means that anybody with a public key *K* can create a namespace rooted in that key. Such a namespace is the most important part of a security domain. Because of (S2), *K* speaks for the domain. Because of (S3), if you know $K^{-1}$ you can delegate authority over any part of the domain, and since *K* is public, anyone can verify these delegations. This means that authentication can happen independent of association with any domain controller. Of course, you can also rely on a third party such as a domain controller to do it for you, and this is necessary if *K* is a symmetric key.

For example, an application such as SQL Server can create its own domain of objects, IDs, names and authorities that has no elements in common with the Windows domain of objects, IDs, names and authorities for the machine on which SQL Server is running. However, the SQL Server can use part or all the Windows security domain if that is desired. That use is controlled by policy, in the form of trust root contents and issued tokens.

Here are some other examples of operating in multiple security domains:

1. A user takes a work laptop home and connects to the home network, which has no connection to the work security domain.
2. A consultant has a laptop that is used in working with two competing companies. For each company, the consultant has a virtual machine with its own virtual disk. Each of those virtual machines joins the Windows domain of its respective company. The host OS, however, is managed by the consultant and has its own local domain.

Sometimes we distinguish between resource domains and account domains, depending on whether the domain mostly contains resources or objects, or mostly contains users or subjects.

Domains can be nested. A child domain has its own management, but can also be managed by its parent.

#### *4.4.2. Attributes*

An attribute such as age=32 is a special kind of path, and thus is a principal like any other. This one has two components, the *name* age and the *value* 32; they are separated

by "=" rather than "/" to emphasize the idea that 32 is a value for the attribute name age, but this is purely syntactic. [7] The claim Paul ⇒ age=32 expresses the fact that Paul has the attribute age=32. Like any path, an attribute should be global if it is to be passed between machines: $K_{oasis}$/age=32. However, unlike file names or people, we expect that most attributes with the same name in many different namespaces will have the same intended meaning in all of them. A claim can translate the attribute from one namespace to another. For example, WA/dmv/age ⇒ NY/rmv/age means that New York trusts WA/dmv for the age attribute. Translation can involve intermediaries: WA/dmv/age ⇒ US/age and US/age ⇒ NY/rmv/age means that New York trusts US for age, and US in turn trusts Washington (presumably US trusts lots of other states as well, but these claims don't say anything about that). Locally, of course, it's fine to use age=32; it's a local name, and if you want to translate US/age=32 to age=32 you need a trust root entry US/age ⇒ age. In fact, from the point of view of trust age=32 is just like a nickname. The difference is that we expect lots of translations, because we expect lots of principals to agree about the meaning of age, whereas we don't expect wide agreement about the meaning of Bob.

Because of the broad scope of many attribute names such as age, the name of an attribute can change as it is expressed in different languages and even different scripts. Therefore it is often necessary to use an ID rather than a name for the attribute in policy. For example, an X.509 object identifier or OID is such an ID. Sections 4.5 and 4.6 discuss the implications of this; what they say applies to attributes as well.

A Boolean-valued attribute (one with a value that is true or false), such as over21, defines a *group*; we normally write it that way rather than as over21=true. The next section discusses groups.

### *4.4.3. Groups and conditions*

A condition is a Boolean expression over attribute names and values, such as "microsoft-.com/division == 'sales' & microsoft.com/region == 'NW'". A condition is a principal; every principal that speaks for attributes whose values cause the expression to evaluate "true" speaks for the condition. In the preceding example, every Microsoft employee in the northwest sales region would speak for it.

For use in conditions, identifiers are considered to be Boolean-valued attributes that evaluate true for the principals that speak for them. Hence the condition paul@micro-soft.com | carl@microsoft.com is true for paul@microsoft.com and carl@microsoft.com. It is also true for the key K if K => paul@microsoft.com.

In addition, there are special attributes, such as time, that may be used in conditions; every principal is considered to speak for them. For example, "time >= 0900 & time <= 1500 & shift == 'day' & jobtitle == 'operator'" would be true for all day-shift operators between 9am and 5pm.

If *C* is a condition, and a principal *P* has attributes whose values cause *C* to evaluate true, then we write:

$P \Rightarrow C$

[7] Sometimes people call age=32 an "attribute-value" or an "attribute-value pair", and call age an "attribute". This is perfectly good English; it might even be better English than calling age=32 an attribute. But it is confusing to have both meanings for "attribute" floating around. In this paper, "attribute" means the pair age=32, and age is the attribute name. Sometimes we say "the age attribute", meaning an attribute whose name is age.

We can give a condition an identifier (a name or an ID) by saying that the condition speaks for the identifier:

$C \Rightarrow identifier$

We call such an identifier a *group*. [8] A group is thus a principal with zero or more other principals that speak for it. If a principal speaks for the group, we say that it is a *member* of the group. Today's groups are defined by a condition that is just the "or" of a list of members. In such a case, it's possible to provide a complete list of all the group members, but this is not always true. The distinction is important for a principal with the authority to define members, but it is invisible to access control, which only cares about a requestor $P$ presenting a claim $P \Rightarrow G$ and $G \Rightarrow resource$ being on the ACL.

Such an authority will only issue such a claim if it:

- Has access to a complete list of the group members (such as Paul, Carl, Charlie), and $P$ is in it, or
- Has access to a partial list of the group members and $P$ is on its partial list; there may be several such lists, each accessible to a different issuer, or
- Knows that $P$ satisfies the condition that defines the group (such as age>=21).

The question of who is trusted to assert $P \Rightarrow G$, that is, who can define the members of a group, is part of authorization.

### 4.5. Global identifiers

To avoid confusion, identifiers communicated between computer systems should be global. If a set of systems doesn't communicate with the rest of the world, they only need to agree among each other. However, when these systems suddenly do need to share identifiers (perhaps because they merge with another set of systems), collisions of identifiers can occur, requiring a massive renaming of entities. To avoid such problems, all identifiers that might travel between computers should be global, except perhaps names intended to communicate to a human being.

An identifier is global if everyone agrees on its meaning, that is, when presented with a request and some supporting evidence, everyone either agrees on whether the identifier is the principal that made the request or doesn't know. A key or hash is automatically global; cryptography makes it so. Other identifiers are paths (perhaps of length one).

A path rooted in a key, such as $K_{intel}$/andy@intel.com, is called *fully qualified*. Such identifiers are global, because $K_{intel}$ is global, and according to rule (S2) above it can say what other keys can speak for identifiers rooted in itself. For example, $K_{intel}$ can establish that Andy's key $K_{andy}$ speaks for the name $K_{intel}$/andy@intel.com, by signing a certificate (token):

(C1) $K_{intel}$ **says** $K_{andy} \Rightarrow K_{intel}$/andy@intel.com

Paths not rooted in keys are rooted in self, the local environment interpreting the identifier. They are not global and therefore should not be sent outside the local environment.

We would like to treat an identifier like andy@intel.com (or /com/intel/andy in the canonical form) as global, even though it is not rooted in a key, because we want to keep keys out of most policy. This is a *conventionally* global identifier: we make it very likely that almost everybody agrees about what speaks for it, by making it very likely

[8] This is not the only meaning of 'group' in English, in computing, or in security, but it is the usual meaning and the one we adopt.

that everyone agrees that $K_{andy} \Rightarrow$ andy@intel.com. We do that by getting the same agreement that $K_{intel} \Rightarrow$ intel.com; then everyone will accept $K_{intel}$'s certificate (C1). Of course this is the same problem, and we can solve it in the same way: agree that $K_{verisign} \Rightarrow$ com, and get a certificate

(C2) $K_{verisign}$ **says** $K_{intel} \Rightarrow$ intel.com

This recursion has to stop somewhere, and it stops in a special part of the security policy called the *trust root*, where some of these facts are built in. The essential idea is:

**Provided their trust roots agree and they have the same tokens, two parties will agree on what keys speak for a conventionally global identifier.**

One case in which the parties might disagree is while a key is being rolled over or replaced, but only if they have different tokens—one has heard about the key change and the other one hasn't.

Section 5.1 discusses the trust root in detail, and section 5.1.1 explains how to make it likely that two trust roots agree.

Although any kind of path could be a conventionally global identifier, the ones that people cares most about are DNS names (see section 4.7). Email names are important too, but they usually don't require special attention because there's a single DNS name that authenticates a given email name.

### *4.6. Choosing identifiers for access policy*

There are three conflicting requirements on identifiers:

- *Meaningful* (to humans): When security policy such as group definitions, access control lists, etc is displayed to humans, identifiers must be meaningful, since people must be able to understand the policy. Only names are meaningful. Another consequence is that only names are controversial: no one cares what bit pattern your public key has, or what domain ID your SID uses, but people do care who controls microsoft.com or mit.edu.
- *Long-lived*: The identifier doesn't need to change when encryption keys or names change. This is desirable, because much security policy is long-lived: the identifier may appear on ACLs for objects that last for decades, and that are scattered over the internet or written on DVDs. Neither names nor direct identifiers can be guaranteed to be long-lived, since people get married, join a new organization, or otherwise change their minds about names, and keys can be compromised and need to change.
- *Direct*: some identifiers must be direct, since only direct identifiers can actually make requests. Direct identifiers are neither meaningful nor long-lived.[9]

The following table summarizes the choices:

| **Property** / **Identifier type** | Meaningful | Long-lived | Direct |
|---|---|---|---|
| Name | **Yes** | no | no |
| ID | No | **possibly** | no |
| Direct (keys, etc.) | No | no | **yes** |

[9] The hash of some data is long-lived in the sense that it won't change. However, the hashes that are important for access control are hashes of code, and the hash of code that you care about changes frequently, because of patches and new versions. So in practice a hash has a much shorter lifetime than many keys.

We can distinguish three main places where an identifier may appear:

- As the direct source of a request, where it must be direct, since all the machine directly knows about the source of a request is the channel it arrives on.
- In the user's view of access control policy, where it must be meaningful, in other words, a name.
- In access control policy stored in the system, where it's desirable for it to be long-lived, but it could have none of these properties as long as there is extra machinery to make up the lack.

As peer-to-peer operation grows—both personal P2P and corporate P2P—identifiers for principals will show up in access control policy far and wide. An identifier might be on ACLs on machines and DVDs all over the world, with no record of where those machines are. It might also be in tokens such as XrML licenses, SAML or XACML tokens, certificates in various forms, etc., which are another way to express access control policy. These signed statements can be carried anywhere, can be backed up, can be transferred from one machine to another. Again, there is no requirement that each such statement have its location registered in any central place. Hence it's often desirable for the identifiers in access control policy to be long-lived.

Since no identifiers satisfy all the requirements, there have to be ways of mapping among them:

- When a request or a token comes in, it can only be authenticated as coming from a direct principal, that is, a channel $C$, so there must be a mapping $C \Rightarrow P$ to a stored principal.
- When a user wants to examine or edit policy they need to see a meaningful principal $M$, so there must be mappings in both directions $M \Rightarrow P$ and $P \Rightarrow M$.

Any kind of identifier can appear in stored access control policy. As we have seen, however, it's often important for stored identifiers to be long lived, so that the policy doesn't have to change when the identifiers change. It's therefore advantageous to use a particular kind of ID called a SID for stored policy, because SIDs are carefully constructed to be long-lived; see section 4.7. There has to be a reliable correspondence between SIDs and names so that policy can be read and written by people, but this correspondence can change with time. There also has to be a reliable SID↔key correspondence so that requests can get access. [10]

The preferred approach to keys is complementary to this one: the only long-term place to store keys should be the trust root (see section 5.1), which contains facts about principals that are installed manually and accepted on faith in reasoning about authentication.

[10]Preferring names would also work, and it would be simpler since there would be no need for the SID↔name correspondence, but it leads to inconvenience when a name changes, and to insecurity when a name is reused.

Preferring keys seems appealing at first, since although it needs a key↔name correspondence, it doesn't need anything else. Unfortunately, it's insecure when a key is compromised, unless the key in policy is no longer treated as a direct identifier but rather as something that can be mapped reliably to a key that is currently valid. Doing this makes it harder to handle than a SID. Since you can't tell by looking at it whether a key has been compromised, you have to do this work every time.

### *4.6.1. Anonymity*

Sometimes people want to avoid using the same identifier for all their interactions with the world, because they want to preserve their anonymity. A variation on this is that they don't want their actions at one web site, for example, to be correlated with their actions at another site; this kind of correlation is called tracking.

Since there is no shortage of encryption keys or identifiers, it's easy for a computer to generate as many identifiers for me as I want, for example, a different one for every web site I interact with. The computer can keep track of which identifier to use at which site. If you are really paranoid, you can use a different identifier each time you go to the *same* site.

In many case, this by itself is sufficient. Sometimes, however, a web site or other party may want to know something about me: that I am over 18, or have a decent credit rating, or whatever. For this purpose a mutually trusted third party such as Live or *Consumer Reports* can authenticate one of my identifiers, certifying, for example, $K_{bwl-amazon} \Rightarrow$ over18. The protocol for this is simple: I authenticate to Live, I give $K_{bwl-amazon}$ to Live and ask for a certificate, and I get back $K_{live}$ **says** $K_{bwl-amazon} \Rightarrow$ over18.

### *4.7. SIDs*

SIDs contain a 96 bit domain identifier plus a 32 bit relative identifier within the domain. Thus the structure is D/R. To distinguish SIDs from other identifiers we prefix SID, so the full identifier is SID/D/R, but we will usually omit the SID/ prefix here. Roughly speaking, D corresponds to something like microsoft.com, and R to blampson or the server red-msg-70, so D/R corresponds to blampson@microsoft.com or red-msg-70.microsoft.com.

These SIDs have the following useful properties:

1. They are not meaningful to humans, unlike names. No one will care which numbers are assigned to which domains or which principals.
2. They are not direct identifiers, as keys are, so that policy expressed in terms of SIDs remains the same when keys change. Only the SID↔key correspondence needs to change.
3. There are plenty of them, so they don't have to be rationed (except to prevent denial of service attacks on ID services that map SIDs to keys).
4. They are (two part) paths D/R, so that a key that speaks for a domain D can speak for lots of SIDs in that domain.

Because of (1) and (2) a SID is a long-lived identifier that is suitable for long-lived policy such as ACLs.

Since there are plenty of domain identifiers, you can get a new one just by choosing a 96 bit random number; this is reasonable because one D is as good as another. The chance of an accidental collision is very small (once every 8,000 years if there are a thousand new domains per second); we consider collisions caused by malice shortly. Some domains will have only a few SIDs (that is, a few values of *R* for one *D*), for example, a domain for a person, family, or small organization. But most SIDs will probably be in large domains belonging to corporations or to Internet services such as Live or Yahoo.

As we saw in the previous section, we need to know $K \Rightarrow D/R$ so that we can authenticate a statement signed by *K* as coming from *D/R*. We also need to know *name* $\Rightarrow$

*D/R* and *D/R* $\Rightarrow$ *name* so that users can read and change policy that is stored in terms of SIDs. These mappings could be strictly local if the local administrator takes responsibility for setting up and maintaining them, but in general it will come from someone who speaks for *D/R* (for example, someone who speaks for *D*) or for name (for example, microsoft.com if *name* is billg@microsoft.com).

Note that joining a Windows domain is quite different from learning $K_D \Rightarrow D$. A machine can only be joined to one domain, and a domain joined machine trusts its domain controller for *any* SID, and also for various management functions. A machine or session can know about lots of domains, and it trusts each one *only* for its own SIDs.

### *4.7.1. Domain ID service*

To simplify the handling of domain key changes and malicious (as opposed to accidental) conflicts for domain identifiers, it's desirable to have one or more domain ID services, which are intended to issue tokens $K_{DR}$ **says** $K_D \Rightarrow D$. Then instead of having a trust root entry for each *D* that you encounter, you only need one that says $K_{DR} \Rightarrow$ *SID*/* for each ID service that you want to trust. For greater security, you could configure your trust root with *n* domain ID services and a requirement that *k* of them agree on $K_D \Rightarrow D$ before it is believed; see section 5.1.2 for more on this. As with other kinds of trust root entries, an entry $K_D \Rightarrow D$ for a specific domain takes precedence, or disagreement is referred to the administrator; see section 5.1. For this to work well, there should not be too many ID services and the scope of each one should be wide.

The domain ID service can work as a simple web service with no human operator involvement only because what it records has no intrinsic value. The ID service is designed specifically and only to meet the needs of authentication. It offers only one public query: "Is $K_D \Rightarrow D$ a registered claim?" [11] It is intentionally not a general purpose directory. It is intentionally limited never to become a general purpose directory. Nothing stops people from making more general directories, but those are not domain ID services.

In addition to the query, there is one operation for registering new values of *D*. The input parameters are *D*, a public key $K_D$, and an optional password *PW* encrypted by $K_{DR}$ that can be used for resetting $K_D$. The request is signed by $K_D^{-1}$. There is no other authentication. In particular, there is no linkage to any PII or to any other information that would require human operators at the domain ID service. After success, $K_D \Rightarrow D$ is a registered claim.

Windows domains today implement a highly simplified version of this scheme, since a domain joined machine trusts its domain controller for any SID.

## *4.8. Names*

The purpose of a name is to be meaningful to a human. Most useful names are paths, and the preferred (conventionally) global names are DNS and email names such as research.microsoft.com or billg@microsoft.com. As we did with SIDs, to distinguish DNS names from other identifiers we prefix DNS, so the full identifier is DNS/com/microsoft/research, but we will usually omit the DNS/ prefix here and use the standard DNS syntax.

[11] Or perhaps "What are the keys that speak for D?"

The crucial security questions about a name are what real world entity it identifies, and what key or SID speaks for it. To answer the second question, you consult the trust root, together with any tokens that are relevant. Thus the trust root might contain

$K_{Verisign} \Rightarrow$ DNS/*; $K_{billg} \Rightarrow$ billg@microsoft.com

Here the second name is written in its conventional email form; as a canonical path name it would be DNS/com/microsoft/email/billg. The rule for trust roots (see section 5.1) is that the more specific entry governs, so that what Verisign or Microsoft have to say about billg@microsoft.com will be ignored.

Today's X.509 trust roots usually grant a certificate authority such as Verisign authority over all DNS names; that is what the $K_{Verisign} \Rightarrow$ DNS/* claim in the example says. Although there are ways to limit the names that such a key can speak for, today they are obscure. Such limits are of fundamental importance, and need to be easy to set and understand.

Adding an entry for a name to the trust root must be a human decision, so the procedure by which the human decides that it's the right thing to do, called a *ceremony*, must be carefully designed. A ceremony is like a network protocol but includes human components as well as computers.

### 4.9. Freshness

Secure communication requires more than assurance that a message came from a known source; it also requires *freshness*, a guarantee that the message is sufficiently recent. Without freshness, a bad guy can make trouble by replaying old messages, which might well be misinterpreted in the current context. For example, consider a request to a service to write a check for $10,000. Replaying this request should not result in a second check. Or consider a request that asks "Does key *K* speak for microsoft.com?" and expects a yes or no answer. If a previous request that asked "Does key $K_{microsoft}$ speak for microsoft.com?" got a "yes" answer, it should not be possible to replay this answer and get the requester to accept it as the answer to the later request.

There are many ways to ensure freshness. In a request-response protocol like the second example above, you tag the request with a sequence number and demand the same sequence number in the response. Such a tag is called a *nonce* or *challenge*. To ensure that an incoming message is fresh, in particular that it was generated since you chose a nonce, you insist that it contain some evidence that the sender received that nonce.

The essential property of a nonce is that it is not reused; nonces may be ordered, but this is usually unimportant. If you want to prevent the responder from precomputing the response, a nonce must be unpredictable; frequently this is not a requirement. Often there are two layers of freshness. For example, a sequence of requests might be carried on a channel that is secured with a fresh key. Then the nonces need only be unique within that sequence, since a different sequence of requests will be secured with a different key. In this example the sequence numbers on the messages don't need to be unpredictable.

To ensure that a key is fresh, generate it by hashing some data that includes a newly generated random number. For two party two-way communication, each party should generate its own random number to be included in the hashed data; this gives each party assurance of freshness, and also ensures a good key even if one of the parties is not good at generating random numbers.

For broadcast communication such as a certificate signed with a public key, nonces don't work because the receivers don't send anything to the broadcaster beforehand.

Instead, we usually rely on a timestamp in the certificate for freshness. The validity period in a token is an example of such a timestamp. You might also want to use a timestamp to avoid a round trip, for instance when sending email. It's not as conclusive as a nonce because of clock skew (and perhaps because it's predictable).

### *4.9.1. Consistency vs. availability*

**Availability and consistency: choose one**

There is a fundamental tradeoff between consistency (or freshness) and availability. *A* is consistent with *B* if *A*'s view of *B*'s state agrees with *B*'s actual state. [12] The only way to ensure this is for *A* to hold a lock on *B*'s state, but this means that *A* has to communicate with *B* to acquire the lock, and after that *B* can't change its state until *A* releases the lock. This is usually unacceptable in a distributed system because it hurts availability too much: if *A* and *B* can't communicate, one of them is going to be stuck. [13]

The alternative is for *A* to settle for a view of *B*'s state at some time in the past; often this is cached information. Now there is a tradeoff among freshness (how far in the past?), availability, and performance (how often does *A* check for changes in *B*'s state?). This tradeoff is fundamental; no cleverness in the implementation can avoid it. The choice is between acting on old (perhaps cached) information, and getting stuck when you can't communicate. This is a management decision and it must be exposed to management control. At least two parameters must be settable by the relying party (perhaps taking account of hints in the token):

1. How old data can be and still be acted on (the tradeoff between freshness and availability).
2. How frequently data should be refreshed (the tradeoff between freshness and performance).

The way to get the freshest information is for *A* to ask *B* for its state right now. This still doesn't guarantee perfect consistency, since *B*'s state can change between the time that *B* sends its reply and the time that *A* receives and acts on it, but it's the best you can do for consistency without a lock. The way to get the greatest availability and the least communication cost is for *A* to act on any view it has of *B*'s state, no matter how old.

This issue shows up most often for authentication in the validity period of a token. A short validity period means that the token is fresh, but also that new tokens must be issued and distributed frequently. A long validity period means that once you have the token you're good to go, but the token's issuer might have changed its mind about the claims in it. Note that there's nothing to stop a relying party from using a different validity period from the one in the token.

### *4.9.2. Revoking claims*

If you have issued a token and you want to cancel it, is there any alternative to letting the validity period expire? Well, yes and no. Yes, because you may be able to revoke the

[12] More precisely, the view is some function *v* of *B*'s state $s_B$, and *A* knows v($s_B^{past}$), where $s_B^{past}$ is some past value of $s_B$. *A* is consistent with *B* if v($s_B^{past}$) = v($s_B$).

[13] Sometimes a special kind of lock called a *lease* is acceptable; this is a lock that times out. A lease prevents its issuer from changing the state until either the leaseholder releases it, or the lease times out. People usually don't use leases for security information, but they could.

token. No, because the revocation is just another kind of token, with a shorter validity period.

The idea behind revocation is that you need two tokens to justify a claim: the original token *Tk* that is "issuer **says** subject $\Rightarrow \ldots$ as long as revoker confirms", and another *confirmation* token "revoker **says** *Tk* is still valid" that has a much shorter validity period than *Tk*. This is better than simply issuing *Tk* with a short validity period because the revoker is optimized for issuing confirmation tokens cheaply, quickly, and with high availability. It can't grant any access by itself, and it doesn't need any detailed information about the principals involved. Its database consists simply of tokens revoked by their issuers. When queried about *Tk*, it checks that database and issues a confirmation token if the database doesn't say that *Tk* is revoked.

To add an entry to the revoker's database, the original issuer writes a token "issuer **says** the token identified by *TkId* has been revoked" and sends it to the revoker. *TkId* could be a hash of the original token or a serial number embedded in the original token. The revoker puts (issuer, *TkId*) in its database. Since issuers can only revoke their own tokens, the revoker doesn't need to know anything about the issuers (unless it wants them to pay). The only harm the revoker can do is to revoke tokens without instructions, that is, mount a denial of service attack.

Because it is much simpler than most issuers and because it can't grant any access by itself, the revoker can afford to issue confirmation tokens with short validity periods, and it can be replicated for high availability. It's important to understand, however, that this is a difference of degree and not of kind. The tradeoffs described in section 4.9.1 still apply; only the parameters are different. For systems that are expected to be connected to the Internet, it's reasonable to use a validity period of a few minutes (or the length of a session, if that is greater). Policy might say that if you can't contact a revoker, you should accept the token anyway.

There are several schemes for revocation. The original X.509 standard specifies a method called a Certificate Revocation List (CRL), but this has fallen out of favor. The revocation scheme usually used for X.509 certificates is the Internet standard OCSP; see [3].

## 5. Authentication

This section describes the core components of authentication, highlighted in Figure 5: the trust root, token sources, and the speaks-for engine. Then it touches briefly on other components: user logon, device and app authentication, compound principals, and capabilities.

Access control is based on checking that the principal making a request is authorized to access the resource, in other words, that the principal speaks for the resource. This check typically involves a trust chain like the one in the example of section 3.2:

$\mathrm{K}_{SSL} \Rightarrow \mathrm{K}_{logon} \Rightarrow \mathrm{K}_{Alice} \Rightarrow$ Alice@Intel $\Rightarrow$ Atom@Microsoft $\Rightarrow_{r/w}$ Spectra

Where do these claims come from? They can be *known*, (that is, built in), or they can be *deduced* from other claims or from tokens, which are claims made by known principals. The trust root holds the built in claims, token sources supply tokens, and the speaks-for engine makes the deductions. Thus these components are the core of authentication:

**Figure 5.** Core authentication components (see section 3.1 for a walk-through)

1. The trust root holds claims that we *know*, such as $K_{Verisign} \Rightarrow$ Verisign. All trust is local, so the trust root is the basis of all trust.
2. Token sources provide claims that *others say*, such as $K_{Verisign}$ **says** $K_{Amazon}$ $\Rightarrow$ Amazon.
3. The speaks-for engine consumes claims and tokens to *deduce* other things we may need to know, such as what tokens to believe, nested group memberships, impersonation, etc.

### *5.1. Trust root*

**All trust is local.**

The trust root is a local store, protected from tampering, that holds things that a system (a machine, a session, an application) *knows* to be true. Everything that a system knows about authentication is based on facts held in its trust root. The trust root needs to be tamper-resistant because attackers who can modify it can assign themselves all the power of any principal allowed on the system.

The trust root is a set of claims (speaks-for facts) that say what keys (or other identifiers) are trusted and what identifiers (names, SIDs) they can speak for. Typical trust root entries are:

| | |
|---|---|
| $K_D \Rightarrow$ SID/$D$ | key $K_D$ speaks for domain identifier $D$ |
| $K_{Microsoft} \Rightarrow$ microsoft.com | key $K_{Microsoft}$ speaks for the name microsoft.com |
| $K_{Verisign} \Rightarrow$ DNS/* | the key $K_{Verisign}$ speaks for any DNS name |
| $K_{DR} \Rightarrow$ SID/* | key $K_{DR}$ speaks for all domain identifiers |

Because all trust is local, the trust root is local, and it must be set up manually. It must also be protected, like any other local store whose integrity is important. Because manual setup is expensive and error-prone, a trust root usually delegates a lot of authority to some third party such as a domain controller or certificate authority. The third claim example above, $K_{Verisign} \Rightarrow$ DNS/*, is such a delegation. It says that Verisign's key is trusted for any DNS name. Another example of such a delegation is the first one above, $K_D \Rightarrow$ SID/$D$, which delegates authority over the domain identifier $D$ to the key $K_D$.

**All trust is partial.**

For convenience people tend to delegate a great deal of authority in the trust root. For example:

- A domain-joined machine trusts its domain controller for any SID.
- Most trust root entries for X.509 certificate authorities trust the authority for any DNS name.
- Today Microsoft Update is trusted by default to change entries in a Windows X.509 trust root.

This is not necessary, however. In a speaks-for claim, a delegation can be as specific as desired. Existing encodings of claims are not completely general, but for example, name constraints in a X.509 certificate can either allow or forbid any set of subtrees of the DNS or email namespace.

A very convenient way of limiting the authority of the delegation in the trust root is the rule that "most specific wins". According to this rule, a trust root with the two entries

$K_{Verisign} \Rightarrow$ DNS/*; $K_{MS} \Rightarrow$ microsoft.com

means that $K_{Verisign}$ speaks for every DNS name except those that start with microsoft.com. It may also be desirable to find out what key $K_{Verisign}$ says speaks for microsoft.com, and notify an administrator if that key is different from $K_{MS}$.

### 5.1.1. *Agreeing on conventionally global identifiers*

As we saw in section 4.5, we would like to use names such as microsoft.com as global identifiers. Since this name doesn't start with a key and therefore is not fully qualified, however, and since all trust is local, this can only be done by convention. There is nothing except convention to stop two different trust roots from trusting two different keys to speak for microsoft.com, or from delegating authority over *.com to two different third parties that have different ideas about what PKI speaks for microsoft.com.

Our goal is that "normal" trust roots should agree on conventionally global identifiers (SIDs and DNS names). We can't force them to agree, but we can encourage them to consult friends, neighbors and recognized authorities, and to compare their contents and notify administrators of any disagreements.

As long as trust roots delegate authority to the same third parties they will agree. If they delegate to two different third parties that agree, the trust roots will also agree.

So it is desirable to systematically detect and report cases where recognized authorities disagree.

### 5.1.2. Replacing keys

The cryptographic mechanisms used in distributed authentication merely take the place, in the digital world, of human authentication processes. These are not just human-scale scenarios performed faster and more accurately, however; they are scenarios that are too complex for unaided humans. Therefore it's important that human intervention be needed as seldom as possible.

It's simple to roll over a cryptographic key automatically, which is fortunate since good cryptographic hygiene demands that this be done at regular intervals. The owner of the old key simply signs a token $K_{old}$ **says** $K_{new} \Rightarrow K_{old}$. Both keys will be valid for some period of time. The main use of these tokens is to persuade each authority that issued a certificate for $K_{old}$ to issue an equivalent certificate for $K_{new}$.

When a cryptographic key is stolen or otherwise compromised, or the corresponding secret key is lost, things are not so simple. If the key is compromised but not lost, often the first step is to revoke it with a revocation certificate $K_{old}$ says "$K_{old}$ is no longer valid"; by a slight extension of (S3), everyone believes this. See section 4.9.2.

The lost or compromised key must now be replaced with a new key. That replacement process requires authentication. In the simplest case, there is an authority responsible for asserting that the key speaks for a SID or name, for example, a trust root (the base case), Verisign or a domain ID service. This authority must have a suitable ceremony for replacing the key. Here are five examples of such a ceremony:

- You sign a replacement request with a backup key.
- You visit the bank in person.
- You give your mother's maiden name.
- You call up your associates in a P2P system on the phone and tell them to change their trust roots.
- Microsoft takes out full page ads in every major newspaper announcing that the Microsoft Update key has been compromised and explaining what you should do to update the trust root of your Windows systems.

## 5.2. Token sources

Recall that a token is a signed claim (speaks-for statement): issuer **says** $P \Rightarrow Q$. In today's Windows, the sources of tokens are highly specialized to particular protocols. For example, a domain controller provides Kerberos tokens, and the SSL protocols obtain server and client certificates. Any entity that obeys a suitable protocol (like the STS protocol for Web Services) can be a source of tokens.

The same host may get tokens from many sources, and any kind of token source can be local, remote, or both. In addition to coming from domain controllers, protocols such as SSL and IPSec, and Web Services Security Token Services, tokens can come from public key certificate authorities, from peer machines, from searches over web pages or online databases that contain tokens, from Personal Trusted Devices such as smart cards or (trusted) cellphones, and from many other places. In corporate scenarios most if not all tokens will probably come from the corporate authentication authority, but in P2P

scenarios they will often come from peer machines as well as from services such as Live. This means that a standard Windows machine needs to be a token source.

The simplest kind of token to manage is signed by a key, and therefore can be stored anywhere since its security depends only on the signature and not on where it is stored. If the token is signed by a public key, anyone can verify it. However, a token can also be signed by a symmetric key, and in this case it usually must come from a trusted online source that shares the symmetric key with the recipient of the token.

### *5.3. Speaks-for engine*

The job of the speaks-for engine is to derive conclusions about what principals are trusted, starting from claims and adding information derived from tokens. The starting claims are:

- The ones in the trust root.
- If you are checking access to a resource that has an ACL, the claims in the ACL. Recall that we view an ACL entry as a claim of the form SID $\Rightarrow_{permissions}$ resource.

Today this reasoning is done in a variety of different places. For example, in Windows:

- Logon, both interactive and network, derives the groups and privileges that a user speaks for; this is called group expansion. Part of this work is done in the host, part in the domain controller.
- X.509 certificate chain validation, which is used to authenticate SSL connections, for example, derives the name that a public key speaks for. In Windows it also does group expansion and optionally maps a certified name to a local account.
- AccessCheck uses an NT token, which asserts that a thread speaks for every SID in a set, and an access control list, which asserts that every SID in a set speaks for a resource, to check that a thread making a request has the necessary access (that is, speaks for) the resource.
- A Web Services STS takes authentication tokens supplied as input and a query, and produces new tokens that match the query. It can do this in any way it likes, but in many cases it has a database that encodes a set of claims (for example, associating keys with users or users with attributes), and the tokens it produces are just the ones that the speaks-for engine would produce from those claims and the inputs.

Although some or all of these specialized reasoning engines may survive for reasons of performance or expediency, or because they implement specialized restrictions, every conclusion about trust should be derived from a set of input claims and tokens using a few simple rules.

The implementation of this tenet is a *speaks-for engine*, a piece of code that takes a set of claims and tokens as input and produces all of the claims that follow from this input. More practically, it produces all of the claims that match some query. In general, the query defines a set of claims. For example, for an access to a resource, the query is "Does this request speak for this resource about this operation". For group expansion, the query is "What are all the groups that this principal speaks for".

The speaks-for engine produces one or more chains of trust demonstrating that principal *P* speaks for resource *T* about access *R*. For example, in section 3 we saw how to demonstrate that $K_{SSL} \Rightarrow_{r/w}$ Spectra by deriving the chain of trust

$K_{SSL} \Rightarrow K_{logon} \Rightarrow K_{Alice} \Rightarrow$ Alice@Intel $\Rightarrow$ Atom@Microsoft $\Rightarrow_{r/w}$ Spectra

Each link in this chain corresponds to a claim, either already in the trust root or derived from a token. For example, we derive $K_{Alice} \Rightarrow$ Alice@Intel.com from the token $K_{Intel}$ **says** $K_{Alice} \Rightarrow$ Alice@Intel.com, using the claim $K_{Intel} \Rightarrow$ Intel.com. This fact comes either from the trust root or from another token $K_{Verisign}$ **says** $K_{Intel} \Rightarrow$ Intel.com, using the claim $K_{Verisign} \Rightarrow$ *.com. So the main chain of trust has auxiliary chains hanging off it to justify the use of tokens. The entire structure forms a *proof tree* for the conclusion $K_{SSL} \Rightarrow_{r/w}$ Spectra.

When *P* is a set of SIDs in an NT token, *R* is a permission expressed in the bit mask form used in Windows and Unix ACLs and *T* has an ACL, this is a very simple, very efficient computational proof.

The full speaks-for calculus extends the flexibility and power of this statement. *P* can be a principal other than SIDs. *T* can be the name of a resource or a named group of resources. Rights *R* can be expressed as names and as named groups of rights. A principal *P* can delegate to *Q* its right *R* to *T* by the token *P* **says** $Q \Rightarrow_R T$ (if *P* has the right to do this).

For example, what can be delegated in an X.509 certificate chain is the permission to speak for some portion of the namespace for which the chain's root key can speak. This does not include the ability to define groups, for example, because group definition is outside the X.509 certificate scope. For that, one can use another encoding of a speaks-for statement (perhaps in SAML, XACML or ISO REL). From the speaks-for engine deduction we can establish that some key (bound to an ID by X.509) speaks for some group (defined by the other encoding—e.g., SAML), and establish that without having to teach SAML to understand X.509 or teach X.509 to understand SAML.

### *5.4. Additional components*

Figure 6 shows all the components of authentication. They are (starting in the lower left corner of the figure and roughly tracing the arrows in the figure, which follow the walkthrough in section 3.1; * marks components already discussed):

1. **User Logon Agent**: a module that is responsible for gathering authentication information from human users.
2. **Logon (in)**: a module that takes logon requests (currently user, network, batch or service), interacts with token sources, and collects the principals that the user speaks for.
3. **Token Sources (User Authentication)**: a source, whether local or remote, such as the Kerberos KDC or an STS, that verifies a logon and provides SIDs or other identifiers to represent the logged-on principal.
4. ***Token Sources (Claims (groups), Token issue)**: a source of group and attribute information. This information may either be obtained over a secure channel, or issued as a token.
5. **Translator**: a dispatcher and a collection of components, each of which verifies the signature on a token and translates that token into an internal claim.

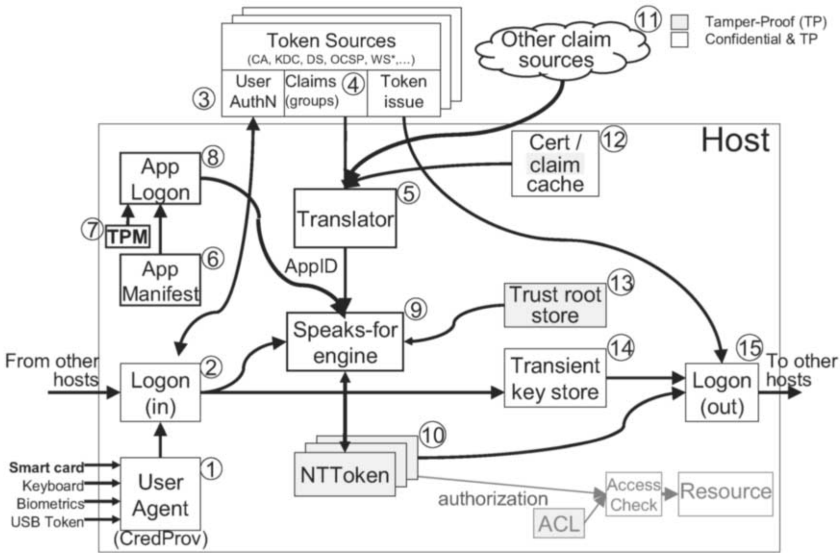



**Figure 6.** Authentication: The full story

6. **App Manifest**: a data structure that completely specifies an application (listing the modules of the application and the hash of each module).
7. **TPM**: hardware support for strong verification of application manifests and of the entire stack on which the application runs.
8. **App Logon**: code that compares an application being loaded into a process against the manifest for that application and, when the two agree, assigns an appropriate SID to that process.
9. ***Speaks-for Engine**: the module that derives claims according to the speaks-for calculus—of primary use in authorization but used in authentication to deduce group memberships.
10. **NT Token**: the existing Windows NT Token—of which there is at least one per session—containing a collection of SIDs identifying the system on which the logon initiated, the user, groups to which this process belongs and the application ID of the process application. In other applications of the architecture this will be a general security context, that is, a principal. Authentication verifies that the user and app speak for this principal.
11. **Other claim sources**: token or claim sources that do not fit the model of Token Sources—tokens or claims can come from anywhere.
12. **Cert / claim cache**: a local cache of certificates or claims (in general, tokens)—in either external or internal form.
13. ***Trust root**: a protected store of speaks-for statements representing things that this session knows.

14. **Transient key store**: a protected and confidential store of cryptographic keys (symmetric keys and private keys) by which this session authenticates (proves) itself to remote entities.
15. **Logon (out)**: the module with which this session authenticates (proves) itself to a remote entity, including both protocols for authentication with negotiation and the user interface that allows a human operator to decide what information to release to the remote system (the CardSpace Identity Selector).

### 5.5. *User agent and logon*

User logon (often called *interactive* logon) does two things:

- It authenticates the user to the host, giving the host evidence that the user is typing on the keyboard and viewing the screen.
- It optionally also makes it possible for the host to convince others that it is acting on behalf of the user without any more user interaction. This process of convincing others is called *network logon*.

There are many subtleties in user authentication that are beyond the scope of this paper. Here are the steps of user authentication in its most straightforward form:

1. The user agent in the host collects some evidence that it interacted with the user, called *credentials*: a nonce signed by a key or password, biometric samples (the output of a biometric reader: measurements of fingerprints, irises, or whatever), a one time password, etc.. Modularity here is for the data collection, which is likely to depend on the type of evidence, and often on the particular hardware device that provides it.
2. It passes this evidence to logon along with the user name.
3. Logon sends the evidence, together with a temporary logon session key $K_{logon}$, over a secure channel to a user authentication service that understands this kind of evidence; the service may be local, like the Windows SAM (Security Accounts Manager), or may be remote (as in the figure) like a domain controller. Modularity here is for the protocol used to communicate with the service. [14]
4. The authentication service evaluates the evidence, and if it is convinced it returns "yes, this evidence speaks for this user name".
5. In addition, to support single sign-on it returns tokens *authority* **says** $K_{logon} \Rightarrow$ *user name* and *authority* **says** $K_{logon} \Rightarrow$ *user SID*. It may also return additional information such as $K_{logon} \Rightarrow$ *authentication method* or $K_{logon} \Rightarrow$ *logon location*.

Single sign-on works by translating the user's interactive authentication to cryptographic authentication. Logon generates a cryptographic key pair for the user's logon

[14] You might think that one protocol could work for any kind of authentication factor. There are two reasons for using different protocols. One is purely historical: existing services used particular protocols. The other is that some protocols, such as Kerberos, depend on the fact that the workstation has a key that it can use to communicate secrets to the service. In Kerberos, for example, the user's password is the source for such a key. Biometric samples don't work. Other protocols, such as SSL, create a secure channel to the service and authenticate it starting with nothing but a trust root entry for a generic authority such as Verisign. As far as I know, SSL secure channel setup together with conventions for finding the service to use, encapsulating the evidence, and allowing for interaction between the user and the service would be a universal protocol.

session. The new key $K_{logon}$ is certified by a more permanent key (on the user's smart-card, in the computer's hardware security module, sealed by a password, from a domain controller, or whatever): $K_{permanent}$ **says** $K_{logon} \Rightarrow$ *user*. It is then used for that one logon session. Since today there are protocols that insist on secret key such as Kerberos, and others that use public key such as SSL, logon should certify one of each.

### 5.6. Device authentication

Device authentication is more subtle than you think. As much as possible, computers and other digital devices should authenticate to each other cryptographically with tokens of the form *K* **says** .... As we have seen, for these to be useful the key *K* must speak for some meaningful name. This section explains how such names get established, using the example of very simple devices such as a light switch or a thermostat. More powerful devices with better I/O, such as PCs, can use the same ideas, but they can be much more chatty.

It is a fundamental fact of cryptographic security that keys must be established initially by some out of band mechanism. There are several ways to do this, but two of them seem practical and are unencumbered by intellectual property restrictions: a pre-assigned meaningful name and a key ferry. This section describes both of them.

You might think that this is a lot of bother over nothing, but consider that lots of wireless microphones and even cameras are likely to be installed in bedrooms in apartments. Some neighbors will certainly be strongly motivated to eavesdrop on these devices. Because the wireless channel is a broadcast channel, the neighbor can mount a "man-in-the-middle" attack that intercepts the messages passing between the device and your computer, and pretends to be the device to the computer and the computer to the device.

#### 5.6.1. Device authentication by name

For device authentication, the simplest such mechanism is for the manufacturer to install a key $K^{-1}$ in the device, give it a name *dn*, and provide a certificate manufacturer says $K \Rightarrow dn$, for example, Honeywell **says** $K \Rightarrow$ thermo524XN12.Honeywell.com. In this example the out of band channel is a piece of paper with the name thermo524XN12 printed on it that comes in the box with the thermostat. After installing the thermostat in the living room, the user goes to a computer, asks it to look around for a new device, reads the name off the screen, compares it with the name on the paper, and assigns the thermostat a meaningful name such as LivingRoomThermostat. Of course a hash of the device's key would do instead of a name, but it may be less meaningful to the user (not that 524XN12 is very meaningful). This protocol only authenticates the device to the computer, not the other way around, but now the computer can "capture" the device by sending it a "only listen to this key" message.

In many important cases this assignment needs to be done only once, even though many different people and computers will interact with the device. For example, a networked projector installed in Microsoft conference room 27/1145 might be given the name projector.27-1145.microsoft.com by the IT department that installs it. When you walk into the conference room and ask your laptop to look around for available projectors, seeing one that can authenticate with that name should be good enough security

for almost anyone. Because this name is very meaningful, authenticating to it is just like authenticating to any other service such as a remote file system.

In many other important cases this assignment only needs to be done very rarely because the device belongs to one computer, which is the device's exclusive user until the computer is replaced. This is typical for an I/O device such as a scanner or keyboard.

### *5.6.2. Device authentication by key ferry*

There are three disadvantages to pre-assigned names that might make you want to use a different scheme:

- You might lose the piece of paper, in which case the device becomes useless.
- You might not trust the manufacturer to assign the name correctly and uniquely.
- You might not trust the user to compare the displayed name with the printed one correctly (or at all, since users like to just click OK)

The alternative to a pre-assigned name as an out of band channel is some sort of physical contact. What makes this problem different from peer-to-peer user authentication is that the device may have very little I/O, and does not have an owner that you can talk to. There are various ways to solve this problem, but the simplest one that doesn't assume a cable or other direct physical connection is a "key ferry". This is a special gadget that can communicate with both host and device using channels that are *physically* secure. This communication can be quite minimal: upload a key from host into ferry at one end; download the key out of ferry into device at the other end. The simplest ferry would plug into USB on the host and the device.

## *5.7. App ID*

This section explains how to authenticate applications. While it's also important to understand how apps are isolated so that it makes sense to hold an app responsible for its requests, this out of scope here.

The basic idea is that apps are principals *just like users*:

- An app is registered in a domain, with an AppSID and a name. This domain is typically the publisher's domain.
- An app is authenticated by the hash of a binary image, just as a user is authenticated by a key.
- When a host makes a new execution environment (process, app domain, etc.) and loads a binary image into it, the new environment gets the hash of the image (and everything that the hash speaks for) as its ID.
- User, machine, and app identifiers can all appear on ACLs or as group members.

Also like users, apps can be put into groups, but this is even more important for apps than it is for users because groups are the tool for managing multiple versions of apps. Like any group membership, the fact that an app is a member of the group can be recorded in AD, or it can be represented in a certificate that is digitally signed by an appropriate authority. Like groups containing users, groups containing apps can nest to make management easier. For example, the GoodApps group might have members GoodOffice, GoodAcrobat, etc.

AppSIDs are probably assigned from the same space as user, group, and machine SIDs, though frequently the AppSIDs are from a "foreign" domain, that of the software publisher (e.g. Microsoft). The assignment is encoded in a signed certificate (usually in the manifest) that associates the binary image with an AppSID and a name in the publisher's domain.

AppSIDs can also be assigned locally by a domain or machine administrator. This must always be done for locally generated applications, and can be done for third party applications (where the AppSID is assigned as part of some approval process). The application is identified by a hash just as in the published case. The local administrator can sign a manifest just like the publisher, or can define a group locally or in AD.

ACLs list the users, machines, and applications that are allowed to access the resource. Sensitive resources might only be accessible through applications in the GoodApps group. Specialized resources might only be accessible to specific applications (plus things like backup and restore utilities).

### *5.7.1. AppSIDs and versions*

A certificate for an app is a signed statement that says something like "hash 743829 ⇒ MS/Word12.3.1, s-msft-word12.3.1. Applications contain many files; a *manifest* is a data structure that defines the entire contents of the application. The manifest includes hashes of all the component files, and it's the hash of the manifest that defines the app.

The manifest can reference system components that are not distributed with the app (e.g. system .dlls). Such a component is considered to be part the platform on which the app is running, not part of the app; see section 5.7.2, and it is referred to by a name, which need not change if the component is patched. There are many complications having to do with side-by-side execution that are not relevant here; it's the platform's job to ensure that the name gets bound appropriately for both security and compatibility. In this respect an app treats a platform component just like a kernel call.

The way this is normally encoded is that the publisher includes the principals that the app speaks for (such as MS/Word12.3.1, s-msft-word12.3.1) in the manifest, and then simply signs the hash of the manifest. This is just a useful coding trick. Of course, the signer of the manifest (or other app certificate) must be authoritative for the domain of the SID and for the name, just as for any other speaks-for statement.

If the system trusts its file store, it can verify the manifest at install time and cache it. This also covers cases where installation includes updates to registry settings and such.

There may be good reasons not to change AppSIDs with each small version change such as a patch. Changing the AppSID requires updating all policy that references it. Some admins will want to do so; others will not. An admin can avoid having to update lots of policy by adding a level of indirection, defining a group and putting the AppSID for each new version into the group; this gives the admin complete control. Publishers can make the admin's life easier by including multiple AppSIDs in a manifest. For example, the manifest for a version of Word might say that it is Word, Word12, and Word12SP2 as well as Word12.3.1. In SP3, the first two SIDs remain the same. Then Contoso ITG can say MS/Word12, MS/Word11.7.3 ⇒ Contoso/GoodWord. Since all trust is local, the structure of the name space for an app is in the end up to the administrator of the machine that runs it. The job of a publisher like Microsoft is to provide some versions and names that are useful to lots of customers, not to meet every conceivable need.

### 5.7.2. The AppID stack

The only assertions an app can make directly are ones encoded in its manifest. When the app is running it depends on its *host environment* to provide the isolation that is needed for an app identity to make any sense. Typically the host environment is itself hosted, so the entire app identity is actually a stack:

StockChart
IE 7.0.1
Vista + patch44325
Viridian hypervisor + patch7654
MachineSID

At the bottom, the machine gets its identity from a key it holds. Ideally this key is protected by the TPM.

We could describe the identity of the app by hashing together the hashes of all the things below it on the stack, just as we hashed all the files of the app together in the manifest. This is probably not a good idea, however, because if there are ten versions of each level in the stack there will be 100,000 different versions—hard to manage. It's better to manage each level separately.

Access control of course sees the whole stack. Taking account of plausible group memberships, an ACL might say GoodApp **on** GoodOS **on** GoodMachine gets access, where "on" here is an informal operator that makes a single principal out of an app running on a host. This makes it easy for the administrator to decide independently which apps, which OS's, and which machines are good. Going further, the administrator might define GoodApp on GoodOS on GoodMachine ⇒ GoodStuff and just put GoodStuff on ACLs.

Note that the policy for what stacks are acceptable might come from the app rather than user or administrator. The main example of this is DRM, in which some remote service that the app calls, such as the license server, demands some kind of evidence that it is running on a suitably secure hardware and OS. The app's manifest might even declare its requirements, but of course an untrustworthy host could ignore them, so the license server has to check the evidence itself. [15]

When a running program loads some new code into itself (a dll, a macro, etc.), it has a number of options about the appID of the resulting execution environment. It can:

1. Use the new code's appID to decide not to load it at all.
2. Trust the code and keep the same AppID the host had before. This is typically what happens at an extensibility point, or in general when an app calls LoadLibrary.
3. Downgrade its own AppID to reflect less trust in the new code.
4. Sandbox the new code and add another level to the stack. Of course the credibility of the resulting AppID is only as good as the isolation of the sandbox.

ACL entries on the operation of loading code can express this choice. Note that when an app calls CreateProcess, for example, it is not loading new code into itself, but asking its host OS to create a sibling execution environment, and it's the host's job to assign the

[15] The app itself could also demand properties from its host, but since the host has complete control over the app, this demand could not be enforced very securely. Ideally the evidence for the license server is a chain of certificates rooted in the hardware TPM's key.

appID for the new process, which might have different, even greater rights that the app that called CreateProcess.

### 5.8. Compound principals

Simple principals that appear in access control policy are usually human beings, devices or applications. In many cases, two or three of these will actually provide proof (authenticate a request). Today only one principal typically provides proof—either a human being or a computer system. Multiple proofs of origin can be used to strengthen security. One important example of this is combining a user identifier and an appID. There are two main ways this can be done:

1. **Protected subsystem**: access is granted only to the combination of two principals, not to either of them alone—for example, opening of a file for backup can be allowed to a registered backup operator, but only when that operator is also running a registered backup application.
2. **Restricted Process**: the desired access is granted only if each of the two or more principals qualify for that access individually [16] —for example, an applet downloaded from a web page at xyz.com might be allowed to access things on xyz.com but not on the user's local machine, and the user running that applet might have access only to objects that the user and the applet both can access.

These two ways of combining principals correspond to **and** and **or**. The principal billg **and** HeadTrax is billg running the HeadTrax protected subsystem; Windows doesn't currently have a way to add such an appID to a security context. The principal billg **or** MyDoom is billg running the MyDoom virus; in Windows today this is a billg process with a MyDoom restricted token.

A Windows security context (or NT token) is a set of SIDs that defines a principal: the **and** of all those SIDs. This principal can exercise all the power that any of those SIDs can exercise. Thus when a security context makes a request, the interpretation is that each of the SIDs independently makes that request; if any of them is on the resource's ACL, the request is granted. So *security context* **says** *request* is SID1 **says** *request* **and** SID2 **says** *request* …, which is another way of saying that *security context* = SID1 **and** SID2 **and** ….

There are other uses for compound principals made with **and**. Financial institutions often demand what they call dual control: two principals have to make a request in order for it to get access to an object such as a bank account. In speaks-for terms, this is $P_1$ **and** $P_2 \Rightarrow$ object. The method for making long-term keys fault-tolerant described in section 5.1.2 is another example of this, which generalized **and** to *k*-of-*n*.

There are also other uses for compound principals made with **or**. In fact, an ACL is such a principal. It says that ($\text{ACE}_1$ **or** … **or** $\text{ACE}_n$) $\Rightarrow$ object.

### 5.9. Capabilities

A capability for an object is a claim that some principal speaks for the object immediately, without any indirection. A familiar example in operating systems is a file descrip-

[16] This kind of access is provided today in Windows by the *restricted token*, in which one has effectively two NTtokens, one for the user's principals and one for a service ID. AccessCheck is called with each of those tokens and the Boolean results of those calls are then **and**ed.

tor or file handle for an open file. When a process opens the file, the OS checks that it speaks for some principal on the file's ACL, and then creates a handle for the open file. The handle encodes the claim that the process speaks directly for reads and writes of the file, without any further checking; this claim is encoded in the OS data structure for the handle. A capability is thus a *summary* of a trust chain. Usually it has a quite limited period of validity, in order to avoid the need to revoke it if the trust chain becomes invalid.

For a capability to work without a common host such as an OS, it must be in a token of the form *object* **says** $P \Rightarrow$ *object* that the object issues after evaluating a trust chain. Later $P$ can make a request along with this token, and the object will grant access without having to examine the whole chain. Such a token doesn't have to be secret, since it only grants authority to $P$.

## 6. Authorization

The main problem with authorization is management. Products usually have enough raw functionality to express the customer's intent, but there is so much detail to master that ordinary mortals are overwhelmed. The administrator (or user) needs a way to build a **model** of the system that drastically reduces the number of items they need to configure. The model needs to not only handle enterprise level security, but also "scale down" to small businesses and homes where there is no professional IT administrator, to peer-to-peer systems, and to mobile platforms and small devices.

Authorization also needs to be feasible to implement. It needs to **scale up** to the Internet, avoiding algorithms and data structures that only work for intranet-sized systems or that depend on having a single management authority for the whole system. Everything that works locally should work on the Internet. Authorization needs to support **least privilege**, by taking account of application as well as user identity, so that trusted apps can get more privileges and untrusted ones fewer; this must work even though apps come in many versions and are extensible. And it needs to be **efficient**: fast in the common case and reasonable in complex cases, even in a large system; it needs to identify problem cases so that people setting policy can avoid them.

### 6.1. Overview

The underlying semantics of authorization is the notion of "speaks-for": there is a chain of principals, starting with the principal making a request (typically a channel on which the request is transmitted or an encryption key that signs the request) and ending with the resource. For example:

$K_{session} \Rightarrow K_{Paul} \Rightarrow$ Paul@microsoft.com $\Rightarrow$ Zeno@microsoft.com
$\Rightarrow$ http://winsecurity/sites/strategy

We call the part of this chain closer to the user "authentication", and the part closer to the resource "authorization". This division is somewhat arbitrary, since there is no sharp dividing line.

In order to make authorization more manageable, you can build a model that collects resources into *scopes* and defines *roles*, each with a set of predefined permissions to execute operations on the resources in the scope. In addition, you can build a *template*

for a scope and its roles, and then instantiate the template multiple times for different collections of resources that have the same pattern of authorization policy. Figure 7 is an overview that shows the main steps in specifying and checking authorization.

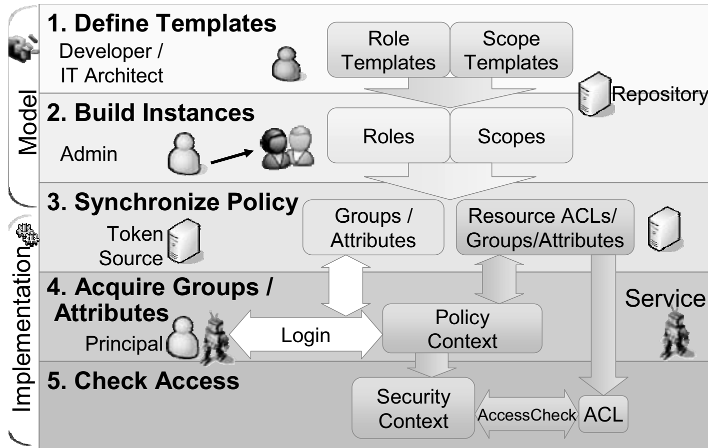



**Figure 7.** Authentication architecture overview.

This model-based access control (MBAC) organizes resources into scopes and principals making requests into roles.

1. The developer or IT architect defines *templates* for scopes and roles that can be used repeatedly in similar situations.
2. The administrator or owner makes *instances* of these templates, groups resources into scopes, and assigns principals to roles.

The remainder of the picture shows how to implement the policy that the model defines.

3. The system compiles or *synchronizes* the model's policy into groups, claims, and ACLs on resources used to do access checks efficiently. When a service starts it acquires its own identity and resource groups, along with those of its enclosing execution environments (OS, device, etc.)
4. The user logs in to a service and acquires groups and claims from the directory or STS to add to the identifiers she already has. The system combines these with resource manager claims and service trust policy to obtain a set of principals that the service thinks the user speaks for.
5. Finally, the set of principals is checked against the ACL for the resource the user is trying to access.

The templates and instances are part of MBAC. The acquisition and access check are part of implementation. The model and implementation are connected when the policy is synchronized.

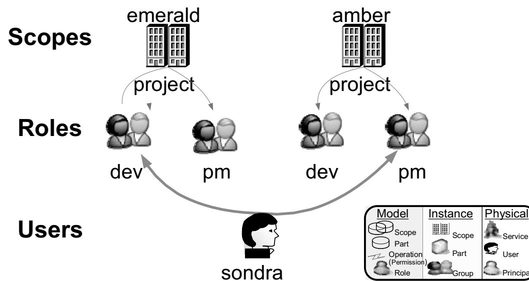



**Figure 8.** The admin sees two scopes, emerald and amber; both are instances of a project repository template. A project has two roles, dev and pm. Sondra is a dev for emerald and a pm for amber.

### 6.2. Model-Based Access Control (MBAC)

The idea of MBAC is to make authorization policy accessible to ordinary mortals; think of it as Excel for authorization. The main customer pain point is that security management is too hard. There are thousands of security knobs (individual ACLs, privileges, resource names, etc.) on each computer, and in a large installation there are thousands of computers. No human can keep that number of separate objects in mind. The model conceals the complexity of the underlying implementation from users and administrators (though they can dive down into individual groups and ACLs if they really need to).

MBAC shines when complex policies apply to multiple objects. It reduces repetitive manual effort by the administrator, and makes it easy to find out what the policy is after a long history of incremental changes. Our examples are necessarily contrived, since something simple enough to put in this paper is simple enough to do manually. So use your imagination to see how the reduction in administrative work is actually substantial for real world scenarios.

Figure 8 shows the administrator's view of a model for part of a system-two project repositories that are scopes for resources, one for the emerald project and one for the amber project. Each project has two roles: one for PMs and one for devs. When deploying a project repository you create a group for each role, containing the users who are in that role for that project. Thus a scope is a collection of resources, and a role is a collection of principals.

This is a simple model-the admin just puts a user, such a Sondra, into the correct group, and all the permissions and memberships are created as a consequence. The actual situation might be messier, as Figure 9 shows. Administering this manually would be quite difficult, but with MBAC the administrator doesn't have to worry about the mess when configuring authorization policy.

Someone has to worry, of course, and that person is the designer of the template, typically a developer or an IT architect. Figure 10 shows the SharePoint template and the emerald.specs scope that is an instance of it. Such a leaf scope corresponds to an

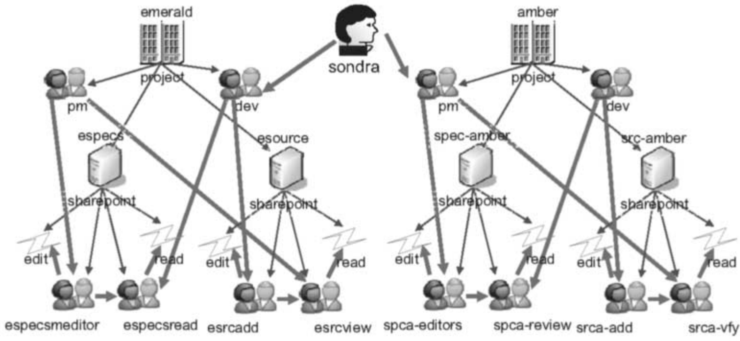



**Figure 9.** Manual administration gets messy

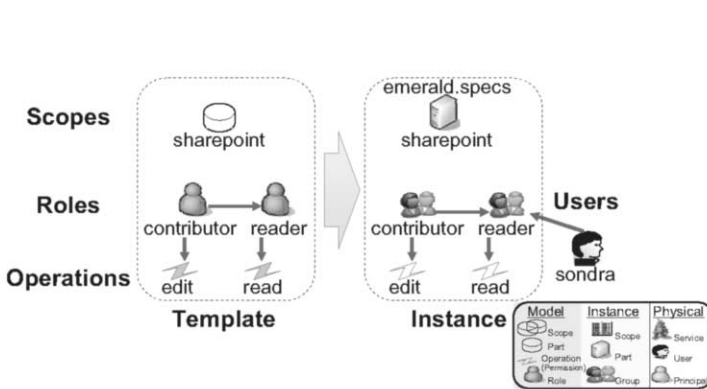



**Figure 10.** A template and an instance emerald.specs for sharepoint; Sondra is a viewer.

instance of a service along with (a subset of) its resources. The developer of the service, in addition to coding the service, creates a **scope template** that defines the roles for the service. A role determines the permissions for a user in that role. Each role is tailored to enable a user to perform some task—like being a teller, or an HR benefits clerk, or in this example, a contributor or viewer of documents on a SharePoint server. A viewer can read documents; a contributor can edit documents, and also is a viewer (this is an example of role nesting). These predefined roles determine the combination of permissions that get tested, to make sure that they correctly enable the desired tasks. Thus the developer or IT architect is responsible for all the details of authorization policy within the scope. From the point of view of the administrator, all the ACLs are immutable.

The administrator instantiates the scope template to create a scope. The same template can be used to create many scopes. Figure 10 shows one of these, in which the

contributor and viewer roles have the same permissions for the SharePoint resource in the scope that the corresponding role templates had in the template. The administrator has put Sondra into the viewer role for the emerald.specs scope. Each scope precisely mirrors the scope template and has the resources, roles, and permissions defined in the template, just as each instance of a class in an object oriented programming language precisely mirrors the class definition.

An IT architect can create higher level templates. In Figure 11 SharePoint is used to create the project repository we described earlier. The project has two subparts, called specs and source. The PM role is assigned to the contributor role in the specs server, and the viewer role in the source server. A part's roles constitute the interface that it exports to containing scopes. The smallest parts are actual services such as SharePoint; composite parts such as project contain subparts. The architect can nest these as deeply as necessary. We expect that there will be a market for templates that are useful to more than one organization.

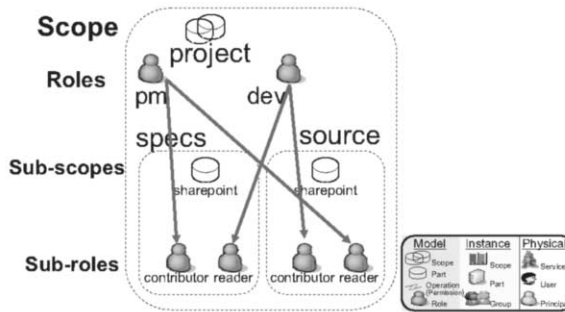



**Figure 11.** Build bigger parts from smaller ones. The specs and source scope templates are SharePoint scope templates that are parts of the outer project scope template, and the inner contributor and viewer role templates are populated from the outer pm and dev ones.

Because the IT architect defines this for all project repositories, all the admin has to do is instantiate the model; she no longer needs to understand all of the details. Two instances of the project template called emerald and amber would get us back to Figure 8.

### *6.3. The model and the real world*

This section explains how the model is connected to the code and data in the real world that it is modeling. Although usually we ignore the distinction between the model and the real world, in this section we need to be clear about it, so we call the real world thing that corresponds to an object in the model its *entity*.

The goal is to keep the model and the real world synchronized, so that changes in entities (and especially creation of new entities) are reflected in the model, and the access control policy set by the model is reflected in its entities. There are three basic issues in synchronization:

1. **Naming**: An object in the model and its entity in the real world are not necessarily named in the same way.
2. **Delay**: An object and its entity are supposed to be in sync, but there may be some delay.
3. **Aggregation**: When entities change, how are the changes aggregated for notifying the model.

*6.3.1. Naming: Paths and handles*

Objects are named by paths: sequences of field names and queries (for selecting an object from a set-valued field). Entities are named by *handles*, which are opaque from the viewpoint of the model. The handle must have enough information to enable secure communication with the root entity.

Because paths and handles are different in general, there has to be a way to map between them. In particular, if the model wants to refer to an object's entity, it needs the entity's handle. Similarly, if an entity wants to refer to its object, it needs the object's path. We take the view that MBAC should work without any changes to entities, as long as they have some sort of interface that is adequate for implementing the get, set, and enum methods described below. Thus the model needs to keep track of each object's handle, which it can do by storing it as part of the object.

In some cases a path may itself be a suitable handle. For example, the model for a file system has objects that correspond to directories and files with isomorphic names. Thus a directory object do has a set-valued contents field whose elements are the files and directories in do, each with a name field. So a file with pathname a\b corresponds to the object whose path is contents?{.name="a"}.contents?{.name="b"}. As this example illustrates, a path may include queries, and hence to use a path as a handle the entities have to be able to understand a query well enough to follow a path. The simplest kind of query has the form [.name = "foo"], where name is a primary key, and this shouldn't be too hard for an entity.

*6.3.2. The model is in charge*

The model can read, and perhaps change, the abstract fields of an entity that correspond to fields of the model by invoking the get and set methods of a corresponding object: obj.get(f) allows the model to read the value of field f in the entity, and obj.set(f, value) allows the model to set the access control policy of the entity. If f is an object, get returns a handle to that object; see below. If a field is a large set, these methods are not suitable, so set fields have a different method: obj.enum(f, i) returns a handle to the ith element of the set, or nil if it has fewer elements (along with a generation number that increases every time something happens to change the object numbering). To change the membership of the set you use operations on the containing scope, such as create. Using these APIs a model can fully explore its entity (as long as the entity isn't changing too fast), learn the handles of all the entities, fill in all the fields of the model, and tell the entity the values of any fields that are determined by the model (normally roles).

In order to use MBAC, an entity must implement these APIs. It may also need to implement query and assign APIs to deal efficiently with large sets of objects. To reflect changes to the entity in the model more efficiently than by polling we may also want a change log. Entries in this log are (h, f) pairs, meaning that field f of entity h has changed.

### 6.3.3. Notification and aggregation

With these APIs the only way for the model to find out about changes in the entities is to do a *crawl*, that is, read out the entire state again with get and enum. This seem impractical for models of any size, so it's necessary to have some kind of change notification. Notification has three issues:

1. It has to be extremely reliable, since if any changes are missed the model's state will diverge from reality, and the only way to get it back in sync is do to a crawl.
2. The entity's name space is handles, so it can only report changes in terms of handles. These have to be mapped to paths.
3. It might be desirable to aggregate all the notifications below some point in the tree.

## 6.4. Scale Up

Current OS authorization mechanisms can scale quite well to enterprises (one Windows AD installation exists that holds 6 million users, for example). They need some work, however, if they are to scale to the Internet, both because things can get much bigger on the Internet, and because there's no single management authority that is universally trusted.

There are some basic features of access control that are important for scaling up:

1. All authentication and authorization statements (speaks-for statements) can be represented in three different ways:

   - They can be stored locally (for example, in the trust root).
   - They can be held in a database on the network (for example, active directory) and delivered over a secure authenticated connection.
   - They can be expressed in a digitally signed certificate (for example, X.509 or SAML tokens), which can be stored and forwarded among the various parties in the transaction.

   The first and third ways permit offline operation and offload of online services (caching). The third way means that claims can be transmitted via untrusted parties.

2. All principal identifiers that are passed from one system to another are globally unique. This means that there's no ambiguity about the meaning of an identifier.
3. Any system or domain can make use of statements from any other domain. It is trust policy, rather than domain boundaries, that distinguishes friend from foe.
4. There is an unavoidable tradeoff among freshness, availability, and performance. If you want the latest information about whether a key is revoked, for example, you cannot proceed if the source of that information is unavailable, and you must pay for the communication to get it. This tradeoff should be controlled by policy, rather than being baked in. For example, here are two possible policies for key revocation:

   - Fail without a fresh OCSP for every access.
   - If OCSP isn't available, treat all cached statements as valid for some period.

Neither one is unconditionally better than the other; it's a matter for administrators' judgment to choose the appropriate one.

In addition to these general principles, there are two topics that require special attention in scaling to the Internet:

- Trust in attribute claims made by other authorities.
- Handling groups, because both the number of groups that a principal belongs to and the total size of a group can become extremely large.

### 6.4.1. Scale Up: Attribute Claims

An attribute differs from a group in two ways:

- It can have a value associated with it, for example, birthdate.
- There may not be a single authority responsible for its definition. For example, birthdates may be certified by any one of 50 state driver's license issuing authorities.

For scaling up, only the second point is important. The first one is handled by conditions.

It is a system's trust policy that handles attributes from other authorities. For example, consider using a driver's license from another state to verify date of birth at a bar in New York. It's convenient for states to agree on the string name of this property. Oasis.org is a standards organization, and we will use oasis.org/birthdate as the standard name.

The first step is for the bar's trust policy to say what the primary authority is for this property:

$K_{NY} \Rightarrow$ oasis.org/birthdate

Then the primary authority says which other sources to trust:

$K_{NY}$ **says** $K_{WA}$/oasis.org/birthdate $\Rightarrow$ oasis.org/birthdate

This says that New York believes Washington about birth dates. If they have a broader agreement, New York might believe Minnesota about all properties defined by oasis.

$K_{NY}$ **says** $K_{MN}$/oasis.org/* $\Rightarrow$ oasis.org/*

Name translation can be done, too. Suppose Illinois doesn't adopt the oasis name:

$K_{NY}$ **says** $K_{IL}$/DOB $\Rightarrow$ oasis.org/birthdate

### 6.4.2. Scale Up: Group Claims

Group membership is a scaling problem today, at least in large organizations. The reason is that a user can be a member of lots of groups, and a group can have lots of members. Today Windows manages this problem in two ways:

- By distinguishing *client* and *resource* groups (also called domain global and domain local groups in Windows), and imposing restrictions on how they can be used.
- By allowing only administrators to define groups used for security.

Figure 12 illustrates the problem. Imagine that ACM creates a group of corporate subscribers to its online digital library. There are 1000 corporate members, each with 10-1,000,000 employees, for a total of millions of individual members. Furthermore, every

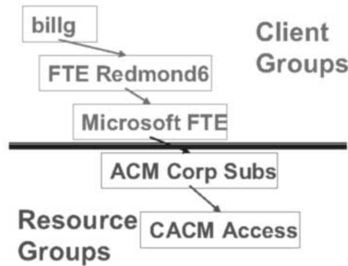



**Figure 12.** Corporate subscribers can access CACM online. The arrows are group membership.

Microsoft employee may implicitly be a member of thousands of such groups, since Microsoft subscribes to lots of services. Thus a client may be in too many groups to list, and a resource may define a group with too many members to list.

In addition, there may be a privacy problem: the client may not want to disclose all its group memberships, and the server may not want to disclose all the groups that it's using for access control.

This is the group expansion, or path discovery, problem. The solution that Windows adopts today, and that we generalize, is to distinguish two kinds of groups:

- **Client** groups (also called *push* groups), which the client is responsible for asserting when it contacts the resource. An individual identifier is a special case of a client group. Thus in Figure 12, the client groups are green: billg, FTE-Redmond6, and MicrosoftFTE. A requestor's client groups are thus known to all resources (subject to privacy constraints), but there can only be a limited number of them.
- **Resource** groups (also called *pull* groups), which the resource is responsible for keeping track of and expanding as far as client group members. In Figure 12 the resource groups are blue: ACMCorpSubs and CACMAccess. The resource thus knows all the client groups that are members, but there can be only a limited number of them.

A client group can only have other client groups as members. This means that there can be only one transition from green to blue in the figure. The client asserts all its client group memberships, and the resource expands its resource groups to the first level of client groups. Consequently, if there is *any* path from the client to the resource, what the client presents and what the resource knows will intersect and the resource will know it should grant access.

Client groups are a generalization of today's domain global groups in AD. Unlike domain global groups, client groups can have members from other domains, but the client *must* know all the client groups it belongs to so that it can assert them, because the resource won't try to expand client groups.

Resource groups are a generalization of today's domain local groups in AD. Unlike domain local groups, resource groups can be listed on the ACL of any resource so long as the resource has permission to read the group membership. It's the resource administrator's job to limit the total size of the group, measured in first-level client groups. The resource *may* cache the membership of third party resource groups.

An added complication is that today Windows eagerly discovers all the resource groups in a domain a client belongs to when the client connects to any resource in the domain. This makes subsequent access checks efficient, and the protocols allow the client and the resource to negotiate at connection time, but if the domain is big (for example, if it contains lots of big file servers) there might be too many resource groups. To handle this, resources may use smaller resource scopes than an entire domain — for example, a service.

To sum up, the way to handle large-scale group expansion is by distinguishing client and resource groups. This extends what Windows does today in five ways:

1. The client and resource can negotiate what group memberships (or other attributes) are needed.
2. Both client and resource can query selected third parties for groups.
3. Both client and resource can cache third party groups. The client must do this, since it must assert all its client groups.
4. The resource can use a smaller scope to limit the number of resource groups that get discovered when the client connects.
5. The client can be configured to know which groups the resource requires.

## References


[1] Abadi and Needham, Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Engineering* **22**, 1 (Jan 1996), 2-15, dlib.computer.org/ts/books/ts1996/pdf/ e0006.pdf or gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-25.html

[2] Internet X.509 Public Key Infrastructure: Certificate and Certificate Revocation List (CRL) Profile, RFC 3280, http://www.ietf.org/rfc/rfc3280.txt

[3] Internet X.509 Public Key Infrastructure: Online Certificate Status Protocol - OCSP, RFC 2560, http://www.ietf.org/rfc/rfc2560.txt

[4] Lampson et al, Authentication in distributed systems: Theory and practice. *ACM Trans. Computer Systems* **10**, 4 (Nov. 1992), pp 265-310, www.acm.org/pubs/citations/ journals/tocs/1992-10-4/p265-lampson

[5] Myers and Liskov, A decentralized model for information flow control, *Proc. 16th ACM Symp. Operating Systems Principles*, Saint-Malo, Oct. 1997, 129-142, www.acm.org/ pubs/citations/proceedings/ops/268998/p129-myers

[6] Wobber et al., Authentication in the Taos operating system. *ACM Trans. Computer Systems* **12**, 1 (Feb. 1994), pp 3-32, www.acm.org/pubs/citations/journals/tocs/1994-12-1/p3-wobber


## A. Basic facts about cryptography

Distributed computer security depends heavily on cryptography, since that is the only practical way to secure communication between two machines that are not in the same room. You can describe cryptography at two levels:

- Concrete: how to manipulate the bits
- Abstract: what the operations are and what properties they have

This section explains abstract cryptography; you can take it on faith that there are concrete ways to implement the abstraction, and that only experts need to know the details.

Cryptography depends on keys. The essential idea is that if you don't know the key, you can't do X, for various values of X. The key is the only thing that is secret; everything about the algorithms and protocols is public. There are two basic kinds of cryptography: public key (for example, RSA or elliptic curve) and symmetric (for example, RC4, DES, or AES). In public key (sometimes called asymmetric) cryptography, keys come in pairs, a *public* key $K$ and a *secret* key $K^{-1}$. The public key is public, and the secret key is the only thing that is kept secret. In symmetric crypto there is only one key, so $K = K^{-1}$.

Cryptography is useful for two things: signing and sealing. Signing provides integrity: an assurance that signed data hasn't changed since it was signed. Sealing provides secrecy: only the intended recipients can learn any of the bits of the original data even if anyone can see all the bits of the sealed data.

For signing, the primitives are Sign($K^{-1}$, *data*), which returns a signature, and Verify($K$, *data*, *signature*), which returns true if and only if *signature* = Sign($K^{-1}$, *data*). The essential property is that to make a signature that verifies with $K$ requires knowing $K^{-1}$, so if you verify a signature, you know it was made by someone that knew $K^{-1}$. With public key, you can verify without being able to sign, and everyone can know K, so the signature is like a network broadcast. With symmetric crypto, anyone who can verify can also sign, since $K = K^{-1}$, so the signature is basically from one signer to one verifier, and there's no way for the verifier to prove just from the signature that the signature came from the signer rather than from the verifier itself.

For sealing, the primitives are Seal($K$, *data*), which returns sealed data, and Unseal($K^{-1}$, *sealedData*), which returns data if and only if *sealedData* = Seal(K, data). The essential property is that you can't learn any bits of *data* (other than its length) from *sealedData* unless you know $K^{-1}$. With public key, anyone can seal data with $K$ (since $K$ is public) so that only one party can unseal it; thus lots of people can send different secrets to the same place. With symmetric crypto, the sealing is basically from one sealer to one unsealer.

There's a trick that uses public key sealing to get the effect of a signature in one important case; it's the usual way of using a certificate to authenticate an SSL session. Suppose you have made up a symmetric key $K$ (usually a session key) and you want to know $K \Rightarrow P$, That is, any messages signed with $K$ that you don't sign yourself come from another party $P$. Suppose you have a certificate for $P$, that is, you know $K_P \Rightarrow P$. This means that only $P$ knows $K^{-1}$. The usual way to authenticate $K$ is to get a signed statement $K_P$ **says** $K \Rightarrow P$ from $P$. Instead, you can compute $SK$ = Seal($K_P$, $K$) and send it to $P$ in the clear. Only $P$ can unseal $SK$, so only $P$ (and you) can know $K$.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*

# Engineering Requirements for System Reliability and Security

Axel van LAMSWEERDE
*Université catholique de Louvain*
*B-1348 Louvain-la-Neuve*
*avl@info.ucl.ac.be*

**Abstract.** Requirements engineering (RE) is concerned with the elicitation of the objectives to be achieved by the system-to-be, the operationalization of such objectives into specifications of requirements and assumptions, the assignment of responsibilities for those specifications to agents such as humans, devices and software, and the evolution of such requirements over time and across system families. Getting high-quality requirements is difficult and critical. Poor requirements were recurrently recognized to be the major cause of system failures. The consequences of such failures may be especially harmful in mission-critical systems.
This paper overviews a systematic, goal-oriented approach to requirements engineering for high-assurance systems. The target of this approach is a complete, consistent, adequate, and structured set of software requirements and environment assumptions. The approach is model-based and partly relies on the use of formal methods *when and where needed* for RE-specific tasks, notably, goal refinement and operationalization, analysis of hazards and threats, conflict management, and synthesis of behavior models.

**Keywords.** Requirements engineering, goal refinement, hazard analysis, threat analysis, inconsistency management, model synthesis from scenarios, agent modeling.

## 1. Introduction

Requirements engineering (RE) embodies a wide range of concerns. The objectives to be achieved by the system-to-be must be elicited and analyzed within some organizational or physical context. Such objectives must be operationalized into specifications of services, constraints, and assumptions. The responsibilities for such specifications need to be assigned among the humans, devices, and software forming the system. Requirements emerge from this process as prescriptive assertions on the software-to-be, formulated in the vocabulary of the environment.

The requirements problem has been with us for a long time. Poor requirements were recurrently recognized to be the major cause of project cost overruns, delivery delays, failure to meet expectations, or severe degradations in the environment controlled by the software. In their early empirical study, Bell and Thayer observed that inadequate, inconsistent, incomplete, or ambiguous requirements are numerous and have a critical impact on the quality of the resulting software [Bel76]. Boehm estimated that the late correction of requirements errors could cost up to 200 times as much as correction during

requirements engineering [Boe81]. In his landmark paper on the essence and accidents of software engineering, Brooks stated that "*the hardest single part of building a sofware system is deciding precisely what to build (...) the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements*" [Bro87]. In her study of software errors in NASA's Voyager and Galileo programs, Lutz reported that the primary cause of safety-related faults was errors in functional and interface requirements [Lut93]. More recent studies have confirmed the requirements problem on a much larger scale. A survey over 8000 projects undertaken by 350 US companies revealed that one third of the projects were never completed and one half succeeded only partially, that is, with partial functionalities, major cost overruns, and significant delays. When asked about the causes of such failure executive managers identifed poor requirements as the major source of problems [Sta95]. On the European side, a survey over 3800 organizations in 17 countries similarly concluded that most of the perceived software problems are in the area of requirements specification and requirements management [ESI96].

Requirements engineering is an intrinsically difficult task:

- it covers a wide spectrum of concerns ranging from high-level, strategic objectives to detailed, technical requirements;
- it involves two systems: the system-as-is and the system-to-be - both including software and environment components;
- it involves stakeholders having diverse, partial, and often conflicting concerns;
- it requires hazardous or malicious behaviors in the environment to be anticipated in order to guarantee requirements completeness and system robustness;
- it requires the evaluation of numerous alternative options: alternative refinements of objectives, alternative assignments of responsibilities, alternative resolutions of conflicts, alternative countermeasures to threats, etc.

The RE process must therefore be supported by systematic methods. To be effective a RE method should meet the following requirements.

- The method should be *goal-oriented* in order to ensure that the requirements meet the system's objectives -including security and safety objectives.
- It should be *incremental* and support early analysis of partial models - the later errors such as omissions, inadequacies, inconsistencies, and imprecisions are found, the more costly their repair is.
- The method should be *constructive* in order to provide analyst guidance and ensure high-quality requirements by construction.
- It should be *model-based* to support abstraction from details and specification structuring. The model should integrate the multiple system facets and support a variety of analyses.
- The method should mix declarative and operational styles of specification as needed.
- It should be formal when and where needed, and lightweight for usability in practical situations.

This paper overviews a RE method addressing these objectives. The method, known as *KAOS*, has been developed and refined for more than fifteen years of research, tool development, and experience in multiple industrial projects. (KAOS stands for "Keep All

Objectives Satisfied".) The details on the modeling notations, model building method, and model analysis techniques can be found in [Lam07].

Section 2 introduces a modeling framework that integrates multiple views of the system-to-be: goals and their refinements; hazards and threats to safety and security goals, respectively; conceptual objects which the goals refer to, together with their interrelationships; operations to ensure that the goals are satisfied; agents responsible for the goals, their behaviors, and interaction scenarios. Section 3 outlines how such a multi-view model can be constructed in a systematic way.

Critical model components should be formalized to enable formal reasoning about them. Section 4 briefly reviews some basics of real-time linear temporal logic for specifying goals, domain properties, hazards, and threats; goal-structured pre- and postconditions for specifying operations; and specification patterns for lightweight specification.

The next sections then discuss various formal reasoning techniques to support the following RE-specific tasks:

- refine goals and check the correctness of refinements (Section 5);
- operationalize fine-grained goals into operations and check the correctness of such operationalizations (Section 6);
- analyze safety hazards by generating obstacles to goal satisfaction and resolving them (Section 7);
- analyze security threats by generating malicious plans to break security goals, and countermeasures to address these (Section 8);
- analyze conflicts among stakeholder goals, and resolve them (Section 9);
- generate system behavior models inductively from interaction scenarios and goal specifications (Section 10).

## 2. A multi-view modeling framework for requirements engineering

The multiple facets of the target system are captured through complementary models:

- a goal model interrelates all intentional aspects;
- an object model defines the structural aspects;
- an agent model defines the system components, their interfaces and responsibilities;
- an operation model defines the functional services in relation with the system goals;
- a behavior model captures agent behaviors in terms of interaction scenarios and parallel state machines;
- obstacle and threat models capture unexpected ways of breaking system goals, including security goals, through incidental or malicious behaviors of environment agents.

We briefly review these models sucessively.

### *2.1. Modeling system goals*

A *goal* is a prescriptive statement of intent [Dar93], [Lam00a]. It expresses some objective to be achieved by the system. The latter comprises the software *and* its environment.

For example, *"train doors shall be closed while the train is moving"* is a goal requiring some cooperation among the software train controller and train sensors and actuators.

Unlike goals, *domain properties* are descriptive statements about the environment, for example, *"a train is moving iff its physical speed is non-null"*.

Goals are defined at different levels of abstraction. Higher-level goals capture global, business-specific objectives, e.g., *"50% increase of transportation capacity"*. Lower-level goals capture local, technical objectives, e.g., *"train acceleration commanded every 3 secs"*.

There are different types of goals. *Functional goals* prescribe intended behaviors declaratively, e.g., *"passengers transported to their destination"*. They are used for building operational models such as use cases, state machines, and the like. *Quality goals* (sometimes called "non-functional goals") refer to non-functional concerns such as security, safety, accuracy, usability, performance, cost, or interoperability, in terms of application-specific concepts. Some of the quality goals are *softgoals*; they cannot be established in clear-cut sense. Softgoals capture preferred behaviors; they are used to compare alternative options [Myl92], [Chu00]. Non-soft goals prescribe sets of admissible behaviors.

Goal satisfaction requires agent cooperation. For example, the high-level goal *"safe train transportation"* requires the cooperation of agents such as the software train controller, the train tracking system, the train driver, passengers, etc. An *agent* is an active system component responsible for goal achievement. Agents refer to roles rather than individuals.

The finer-grained a goal is, the fewer agents are required for its satisfaction. A *requirement* is a goal assigned to a single agent in the software-to-be. For example, *"doorState = 'closed' while measured speed is non-zero"* is a requirement on the train controller. An expectation is a goal assigned to a single agent in the software environment. For example, *"passengers exit train when doors are open at their destination"* is an expectation. Expectations are sometimes called assumptions; unlike requirements they cannot be enforced by the software-to-be.

One important, often neglected part of the requirements engineer's job is to provide *satisfaction arguments* [Lam00a], [Ham01]. These take the form

$$R, E, D \vdash G,$$

meaning "in view of properties $D$ of the domain, the requirements $R$ satisfy goal $G$ under expectations $E$".

Goals provide a criterion for requirements completeness and pertinence [Yue87]. Let REQ, EXPECT, and DOM denote a set of requirements, expectations, and domain properties, respectively.

A requirements set REQ is *complete* if for all identified goals G:

$$\{REQ, EXPECT, Dom\} \vdash G$$

A requirement r in REQ is *pertinent* if for some identified goal G:

r is used in a satisfaction argument $\{REQ, EXPECT, Dom\} \vdash G$

Note thus that requirements completeness and pertinence is relative to known domain properties and the identified goals and expectations.

A *goal model* shows contribution links among goals. It is represented by an AND/OR refinement graph whose nodes represent goals and edges represent AND/OR refinement links. In this graph, a goal $G$ is *AND-refined* into subgoals $G1, G2, \ldots, Gn$ iff satisfying $G1, G2, \ldots, Gn$ contributes to satisfying $G$. (A more precise definition is

given in Section 5.) The set {G1, G2, ..., Gn} is called refinement of *G*. A goal *G* is *OR-refined* into refinements *R1, R2, ..., Rm* iff satisfying the subgoals of *Ri* is one alternative to satisfying G ($1 \leq i \leq m$). *Ri* is called an alternative for *G*. Fig. 1 shows a goal model fragment for our train control system.

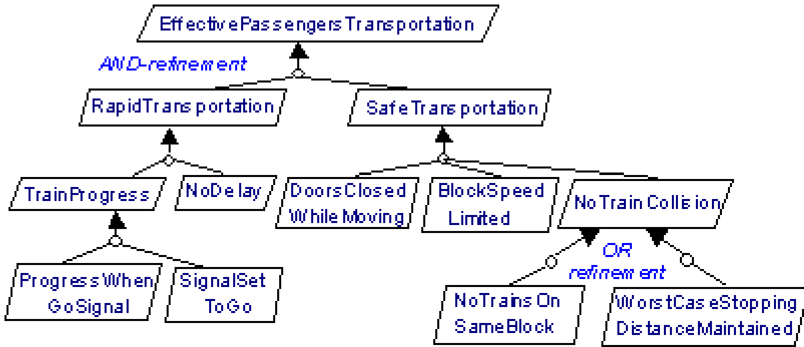



**Figure 1.** Portion of a goal graph in a train control system [Lam07]

Goal models are built using a variety of elicitation techniques. Preliminary goals are identified by analyzing the problems and deficiencies in the system-as-is, and by searching intentional and prescriptive keywords in available raw material and interview transcripts. More abstract, coarse-grained goals are then obtained bottom-up by asking *WHY* questions about available goals and operational material such as scenarios [Jar98]. In parallel, more concrete, fine-grained goals are obtained top-down by asking *HOW* questions about available goals. Goals are also derived by use of refinement patterns (see Section 5), by resolution of obstacles (see Section 7), and by exploration of countermeasures to security threats (see Section 8).

In this model elaboration process, goal refinement terminates when fine-grained subgoals are obtained that can be assigned as requirements or expectations to software or environment agents, respectively [Dar93]. Goal abstraction terminates when the system boundary is reached, that is, the more abstract supergoals cannot be satisfied under the sole responsibility of the agents forming the system.

The nodes in a goal model are decorated by annotations to characterize the corresponding goal - such as its precise definition, an optional formal specification of the goal in a real-time temporal logic (see Section 4), the goal's priority level, etc.

### *2.2. Modeling system objects*

The object model provides a structural view of the target system. A conceptual object is a thing of interest in the system whose instances can be distinctly identified, share similar features, and have a specific behavior from state to state. An object is modeled as an *entity*, *association*, or *event* dependent on whether it is an autonomous, subordinate, or instantaneous object, respectively. The object model is represented by an operation-free, design-independent UML class diagram.

Such diagram can be systematically derived from the goal model [Lam00a]. Each goal formulation is analyzed to extract the entities, associations, and attributes the goal

refers to. For example, a goal *"avoid multiple trains on the same block"* gives rise to *"Train"* and *"Block"* entities and an *"On"* association. The goal *"train speed shall not exceed the speed limit of the block which the train is on"* gives rise to a *"speedLimit"* attribute of *"Block"*, etc.

Contrarily to what is is often confessed in the UML literature, no "hocus pocus" is required here to obtain a "good" object model; goal-directed construction guarantees a complete and pertinent object model.

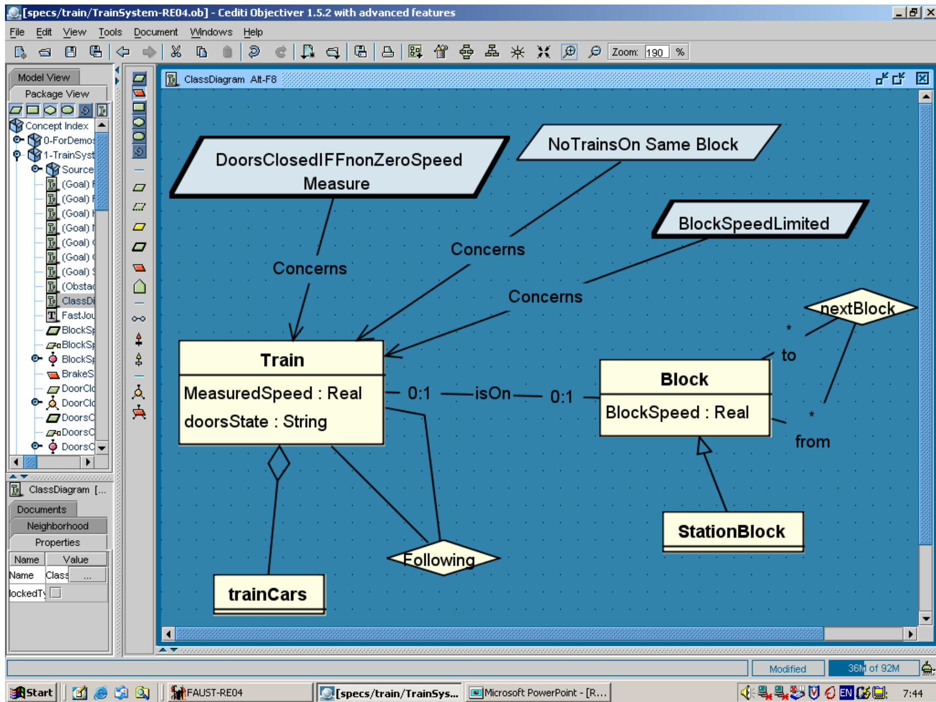



**Figure 2.** Modeling objects referred to by goals [Lam07]

Fig. 2 illustrates an object model fragment for our train control system. The nodes in an object model are decorated by annotations to characterize the corresponding object - such as its precise definition, domain properties associated with the object (that can be optionally specified in real-time temporal logic), etc.

## *2.3. Modeling system agents*

The agent model defines the responsibilities and interfaces of the various agents. As introduced before, an agent is a software, device, or human component of the system that plays some specific role in goal satisfaction. It controls behaviors by performing operations (see Section 2.4). Agents run concurrently with each other.

An agent is modelled by responsibility links to goals and by monitoring/control links to object attributes and/or associations from the object model. Monitoring/control links capture the agent's interface through the state variables it monitors and controls in its

own environment [Par95]. A *state variable* is an attribute or association of some object. Each state variable is controlled by a single agent.

An agent responsible for some goal must restrict system behaviors [Fea87]. The goal must be realizable by the agent [Let02a]. A goal *G* is *realizable* by agent *ag* iff :

- (intuitively:) given ag's *monitoring & control* capabilities it is possible for *ag* alone to satisfy *G* without more restrictions than required by G;
- (more formally:) there exists a transition system $TS_{ag} = (Init, Next)$ on the state variables monitored and controlled by *ag* such that $RUN(TS_{ag}) = HISTORIES(G)$, that is, the set of agent runs equals the set of behaviors prescribed by the goal.

There can be multiple causes for goal unrealizability, namely, (a) lack of monitorability of variables to be evaluated in the goal formulation, (b) lack of controllability of the variables constrained by the goal, (c) need to evaluate variables in future states, (d) conditional goal unsatisfiability, or (e) reference to a target condition to be achieved in unbounded future. This taxonomy of unrealizability problems gives rise to goal refinement tactics for resolving unrealizability [Let02a]. The latter are encoded as refinement patterns (see Section 5).

In an agent model, OR-assignment links allow us to represent alternative assignments of the same goal to different agents. Alternative software-environment boundaries can thereby be captured and assessed with respect to softgoals [Chu00] so as to select a "best" responsibility assignment.

Responsibility assignments also provide a basis for simple forms of load analysis. Fig. 3 shows the responsibilities of an overloaded air traffic controller. This view was generated from a corresponding agent model using a query/visualization tool on the model database.

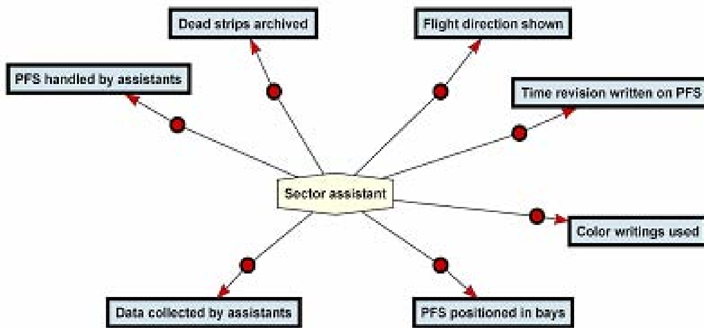



**Figure 3.** Load analysis [Lam07]

## 2.4. Modeling system operations

The operation model provides a functional view of the target system in terms of the services to be provided.

An operation *Op* is a relation: $Op \subseteq InputState \times OutputState$. It must operationalize some underlying goals from the goal model; this entails a proof obligation (see Section 6).

Operation applications yield state transitions and corresponding events. They are atomic; an input state is mapped to a state at next smallest time unit. (Operations with duration are represented through start/end events.) They can be concurrent with others.

In an operation model, operations are connected to goals via operationalization links, to objects via input/output links, and to agents via performance links. UML use case models can easily be generated from such models.

Each operation in an operation model is specified by a pair of conditions (*DomPre*, *DomPost*) where:

- *DomPre* is a *descriptive* condition that fully characterizes the class of input states of the operation in the domain,
- *DomPost* is a *descriptive* condition that fully characterizes the class of output states of the operation in the domain.

An operationalization of a goal *G* into operation *Op* is further specified by a triple of conditions (*ReqPre*, *ReqTrig*, *ReqPost*) where:

- *ReqPre* is a prescriptive necessary condition on *Op*'s input states to ensure *G*;
- *ReqTrig* is a prescriptive sufficient condition on *Op*'s input states to ensure *G*; it requires immediate application of *Op* provided *DomPre* holds;
- *ReqPost* is a prescriptive condition on *Op*'s output states to ensure *G*.

As an operation may contribute to multiple goals, it can have multiple required preconditions, trigger conditions, and/or postconditions. The global precondition for the operation to be applied is

$$Pre = DomPre \wedge \bigwedge_i ReqPre_i$$

The global postcondition when the operation is applied is

$$Post = DomPost \wedge \bigwedge_j ReqPost_j$$

The global trigger condition forcing the operation to be applied is

$$Trig = \bigvee_k ReqTrig_k$$

The specifier must always ensure the following consistency rule:

$$\bigvee_k ReqTrig_k \wedge DomPre \Rightarrow \bigwedge_i ReqPre_i$$

In our train control example, the operation for opening train doors might be specified as follows:

**Operation** OpenDoors
**Def** *Operation controlling the opening of all train doors*
**Input** Train, **Output** Train/DoorsState

**DomPre** *The train doors are closed*
**DomPost** *The train doors are open*
**ReqPre For** DoorsClosedWhileNonZeroSpeed
*The train's measured speed is 0*
**ReqPre For** SafeEntry&Exit
*The train is at some platform*
**ReqTrig For** NoDelayToPassengers
*The train has just stopped*

A corresponding formal version can optionally be specified as well (see Section 5). The distinction between domain and required conditions is important. Unlike in most specification languages, we are not confusing descriptions and prescriptions. Prescriptions may be assessed, negotiated, and replaced by alternatives; descriptions may not. Moreover, *traceability* between operations and their underlying goals is thereby supported.

### 2.5. Modeling obstacles to goals

The goals identified in the early stages of the RE process are often too ideal. They are likely to be violated due to unexpected or malicious agent behaviors. For system robustness and requirements completeness, it is essential to detect and resolve such "overoptimism" at RE time - especially in the case of mission-critical systems.

An obstacle $O$ to goal $G$ is a goal violation precondition satisfying the three following conditions:

1. $\text{O, Dom} \models \neg \text{G}$ *obstruction*
2. $\text{Dom} \not\models \neg \text{O}$ *domain consistency*
3. There exists a behavior $E$ of the environment of the set of agents in charge of $G$ such that $E \models O$ *feasibility*

Hazards and threats are obstacles obstructing safety and security goals, respectively.

An *obstacle model* is a set of goal-anchored fault trees where each fault tree is an AND/OR refinement tree showing how the goal can be violated. The root of the tree is the goal negation; the leaves are elementary obstruction conditions that are consistent with the domain and satisfiable by the environment.

Obstacle resolution then consists in overcoming the sub-obstacles through various resolution tactics such as *goal weakening*, *goal substitution*, *agent substitution*, *obstacle mitigation*, and so forth [Lam00b]. Such resolution yields new or deidealized goals, resulting in a more complete set of requirements for a more robust system.

Fig. 4 shows an obstacle OR-refinement tree showing how the goal "*train stopped if signal set to 'stop'*" could be broken. The new goal of regularly sending out responsiveness checks to train drivers emerges there as a resolution to the sub-obstacle in the middle. The leaf obstacles in Fig. 4 occurred in various reported train accidents (see ACM's Risks forum.)

### 2.6. Modeling security threats

Threat models are augmented obstacle models where:

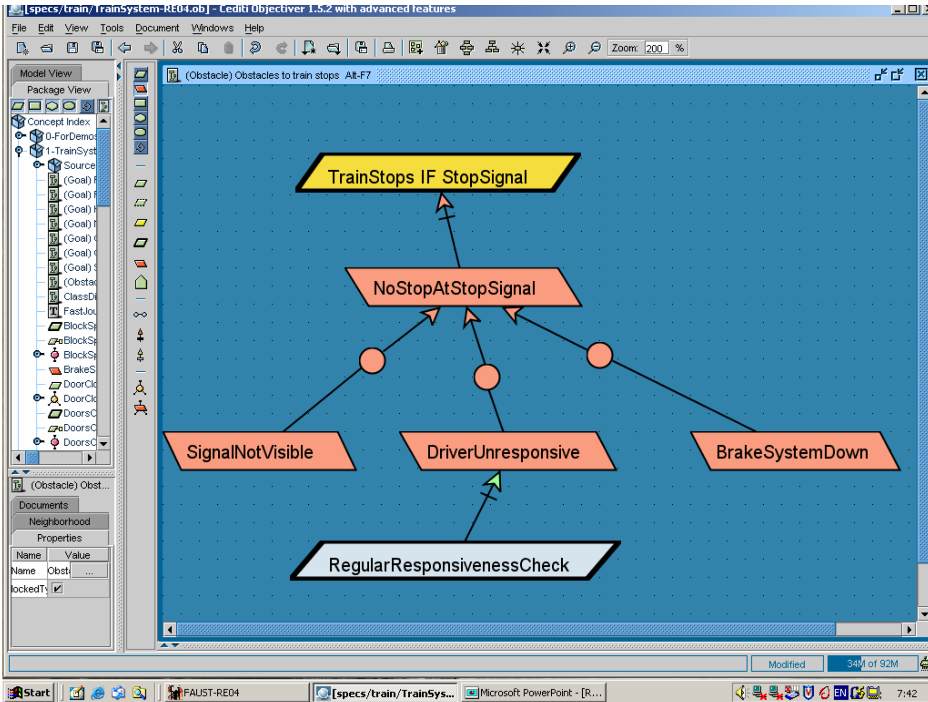



**Figure 4.** Portion of an obstacle model with new goal as resolution [Lam07]

- the root goal negation refers to a security goal,
- the obstacles are malicious obstacles (called *threats*),
- the refinement graph is extended with the attacker's anti-goals,
- the refinement terminates when leaf conditions are reached that can be monitored and controlled by the attacker.

Such models can be built systematically [Lam04a], and even automatically under certain restrictions [Jan06], see Section 8. They provide the basis for enriching the goal model with countermeasures to the identified threats.

### 2.7. *Modeling agent behaviors*

The agent behaviors are modelled by interaction scenarios at instance level and by parallel state machines at class level.

A *scenario* is a historical sequence of interaction events among agent instances. It illustrates some way of achieving a goal G; the scenario is a sub-history in the set of admissible behaviors prescribed by G. An interaction event corresponds to an application of some operation by a source agent, notified to a target agent.

Scenarios can be positive or negative. A *positive* scenario is an example of desired behavior. A *negative* scenario is a counterexample showing some undesired behavior. A scenario may be composed of sub-scenarios, called episodes, which may be common to multiple scenarios.

Scenarios are represented by simple message sequence charts (MSCs), see Fig. 5. Such diagrams capture a partial order on interaction events and, along each agent's timeline, a total order on events.

Scenarios and goals have complementary benefits. Scenarios provide a concrete, narrative way of eliciting requirements from examples and counterexamples. They also give us acceptance test data for free. On the downside, they are inherently partial and raise a coverage problem similar to test cases. They lead to a combinatorial explosion of traces for good coverage. They often entail premature choices such as unnecessary sequencing of events or decisions on the software-environment boundary. Last but not least, they keep the underlying requirements implicit. Scenarios are therefore useful for requirements elicitation and validation whereas goals are required for declarative reasoning (see sections below). Moreover, goal specifications can be inductively synthesized from scenario examples and counterexamples using learning algorithms [Lam98b].

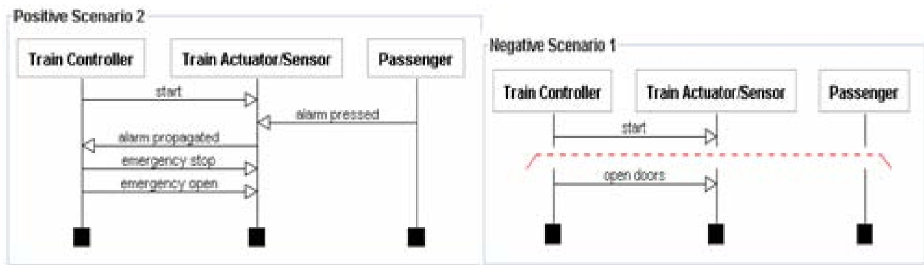



**Figure 5.** Positive and negative scenarios [Lam07]

At class level, the system's behavior is modelled as a parallel composition of agent behaviors. Each agent is behaviorally modeled by a labelled transition system (LTS), see [Mag06]. Agents thereby behave asynchronously but synchronize on shared events. Fig. 6 shows a LTS model that covers all possible event traces for train controllers in the system. Note that the event trace along the train controller's timeline in the positive scenario in Fig. 5 is covered by a path in the LTS model in Fig. 6.

LTS models have a simple, compositional semantics. They are executable for model animation [Mag00]. They support a variety of analyses including model checking [Gia03]. On the downside, they are hard to build and understand. Section 10 will outline a technique for synthesizing them inductively and interactively from scenarios and goals ([Dam05], [Dam06]).

## 3. Building the entire system model: a systematic method

In a model-based requirements engineering process, the various system views outlined in the previous section can be elaborated and integrated in a systematic way through the following general steps.

1. *Domain analysis*. Build a goal model for the current system-as-is through *WHY* and *HOW* questions on available material. (Section 5 outlines techniques to support this step.) In parallel, elicit scenarios of doing things in the system-as-is as illustrations of behaviors prescribed by goals.

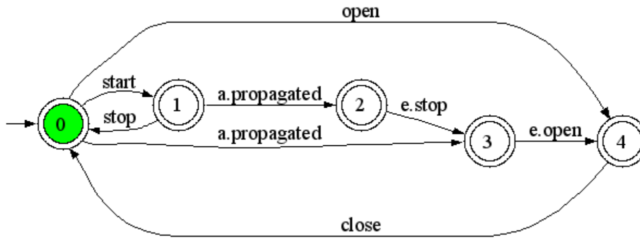



**Figure 6.** LTS model for train controllers [Lam07]

2. *Domain analysis*. Derive an object model for the system-*as-is* from this goal model.
3. *System-to-be analysis*. Replay Step 1 for the system-*to-be*: based on elicited material about the experienced problems with the system-as-is and emerging opportunities, update and expand the previous goal model. The upper levels of the goal model often remain unchanged as organization-wide business objectives tend to be fairly stable. New, specific features of the system-to-be are developed along OR-branches in the goal graph as alternative ways of meeting the same higher-level goals in view of the elicited problems and opportunities. (Section 5 outlines techniques to support this step.) In parallel, explore new scenarios of doing things in the system-to-be as illustrations of behaviors prescribed by new goals.
4. *System-to-be analysis*. Replay Step 2 for the system-*to-be*: update and expand the previous object model from the new version of the goal model. The basic concepts from the application domain often remain unchanged.
5. *Obstacle and threat analysis*. In parallel with Steps 1-4, build obstacle and threat models, and explore resolutions to enrich and update the goal model. (Sections 7 and 8 outline techniques to support this step.)
6. *Conflict analysis*. In parallel with Steps 1-5, detect conflicts among goals and explore resolutions to enrich and update the goal model. (Section 9 outlines techniques to support this step.)
7. *Responsibility* analysis. Explore alternative assignments of leaf goals to system agents, select best alternatives based on non-functional goals from the goal model [Chu00], and build an agent model.
8. *Goal operationalization*. Build an operation model ensuring that all leaf goals from the goal model are satisfied. (Section 6 outlines techniques to support this step.)
9. *Behavior analysis*. Build a behavior model for the system as parallel composition of behavior models for each component. Animate this model for adequacy checking and feeback from stakeholders [Tra04]. (Section 10 presents techniques to support this step.)

The above steps are ordered by data dependencies. They are often intertwined, with backtracking to previous steps. In particular, behavior models are sometimes built earlier in the process, for understanding the system-as-is and for earlier animation of portions of the system-to-be. See [Dar93], [Lam00a], [Lam07] for heuristics, justifications, and

illustrations of each step above. Industrial experience with this method is reported in [Lam04b].

## 4. Formal specification of goals, domain properties, and operations

The approach presented here is a "two-button" one where the formal analysis button is pressed only when and where needed. The button pressed by default is the semi-formal one, where the modeler is using the graphical notations and supporting tools to elaborate her models, perform static semantics checks on them through queries on the model database, generate HTML files for model browsing, generate UML use cases and other derived diagrams, and generate the requirements document [Obj04].

Formal analysis of critical aspects in the models require a formal specification language for the goals, domain properties attached to objects, and pre- and postconditions on the operations. A linear real-time temporal logic (RT-LTL) is used for the goals, domain properties, and required trigger conditions (for the latter conditions, with past operators only). A simple state-based, Z-like language is used for the domain and required pre- and postconditions.

The main temporal operators used are the following standard ones:

- $\circ$ P: P holds in the next state
- $\Box$ P: P holds in every future state
- P ***W*** N: P holds in every future state unless N holds
- $\Diamond$ P: P holds in some future state
- $\Box_{\leq T}$ P: P holds in every future state up to T time units
- $\Diamond_{\leq T}$ P: P holds within T time units
- P $\Rightarrow$ Q for $\Box$ (P $\rightarrow$ Q)

The counterpart over past states is provided by past, "blackened" operators, e.g.,

- $\bullet$ P: P holds in the previous state
- @ P $\bullet (\neg$ P) $\wedge$ P

These formulas are interpreted as usual over historical sequences $H$ of states, e.g.,

$(H, i) \models \Box P$ iff $(H, j) \models P$ for all $j \geq i$

$(H, i) \models \Diamond_{\leq T} P$ iff $(H, j) \models P$ for some $j \geq i$ with dist $(i,j) \leq T$

The $\circ$/$\bullet$ operators refer to the next/previous state within the smallest time unit. They are often used for expressing immediate obligations.

Here are some examples of formal specifications.

**Goal** *Maintain* [DoorsClosedWhileNonZeroSpeed]
  **FormalSpec** $\forall$ tr: Train
    tr.MeasuredSpeed $\neq$ 0 $\Rightarrow$ tr. DoorsState = 'closed'

**Goal** *Achieve* [FastJourney]
  **FormalSpec** $\forall$ tr: Train, bl: Block
    On (tr, bl) $\Rightarrow$ $\Diamond_{\leq T}$ On (tr, next(bl))

In goal specifications, the keywords prefixing goal names are used to indicate temporal specification patterns ([Dar93], [Dwy99]) - e.g., *Achieve [P]* indicates a pattern $\Diamond_{\leq T}$ $P$ on a target predicate P; *Avoid [P]* indicates a pattern $\Box \neg P$; and so forth. Such patterns help writing the specification from informal prescriptive statements. They prove convenient for non-expert specifiers to use elementary temporal logic without knowing it.

The operation of controlling the opening of train doors is formally specified as follows:

**Operation** OpenDoors
**Input** tr: Train; **Output** tr: Train/DoorsState
**DomPre** tr. DoorsState = 'closed'
**DomPost** tr. DoorsState = 'open'
**ReqPre for** *DoorsClosedWhileNonZeroSpeed*: tr.MeasuredSpeed = 0
**ReqPre for** *SafeEntry&Exit*: (∃ pl: Platform) At (tr, pl)
**ReqTrig For** *NoDelayToPassengers*: @(tr.MeasuredSpeed = 0)

The system's semantic picture is as follows. The global state of the system at some time position is the aggregation of the local states of all its agents at that time position. The local state of an agent at some time position is the aggregation of the states, at that time position, of all the state variables the agent controls. (Such variables are attributes and/or associations from the object model, see Section 2.3.) The state of a variable at some time position is a mapping from its name to its value at that time position. The system evolves synchronously from system state to system state, where the time distance between successive states is the smallest time unit defined in the RT-LTL language. (This time unit may be chosen arbitrarily small.)

A system's state transition is caused by the application, by some agents, of applicable operations they may or must perform on the state variables they control. As introduced in Section 2.4, operations are atomic; an operation applied in the current state maps the corresponding agent's state to the next state one smallest time unit later. As multiple trigger conditions may become true in the same state, the corresponding operations *must* fire simultaneously. We thus have true concurrency here; a system's state transition is composed of parallel transitions on local states. An interleaving semantics is not possible in view of the obligations expressed by trigger conditions.

The system's non-determinism arises from the non-deterministic behavior of its agents. While an agent *must* perform an operation when one of the operation's trigger conditions becomes true, the agent has the freedom to perform an operation or not when its required preconditions are all true. Such non-determinism, while suitable at a more abstract level for declarative reasoning, must in general be removed when the specification is translated into a more operational language (e.g., for specification animation or other checks on the operational version) [Del03], [Tra04]. A choice must then be made between an eager or lazy behavior scheme for each operation performed by the agent. In the *eager* behavior scheme, the agent performs the operation as soon as it can, that is, as soon as *all* required *preconditions* are true. This corresponds to a maximal progress property. In the *lazy* behavior scheme, the agent performs the operation when it is really obliged to do so, that is, when *one* of its required *trigger conditions* becomes true.

A system's behavior is then defined by a temporal sequence of system state transitions. The system *satisfies* a non-soft goal if the set of all its possible behaviors is included in the set of behaviors prescribed by the RT-LTL specification of the goal.

As opposed to generative semantics of operational languages such as Statecharts or VDM, where every state transition is forbidden except the ones explicitly required by the specification, we have a *pruning semantics* here: every state transition is allowed except the ones explicitly forbidden by the specification. With a generative semantics, operations are viewed as generating the set of admissible behaviors of the system; these

cover the only possible transitions. As a consequence, generative semantics have a built-in assumption that nothing changes except when an operation specification explicitly requires it; the specifier is relieved from explicitly specifying what does *not* change - in other words, a generative semantics avoids the *frame problem* [Bor93]. Such built-in frame assumption, however, makes it difficult to support incremental reasoning about partial models [Jac95]. With a pruning semantics (like in Z, LARCH, or other temporal logic-based formalisms), the specification prunes the set of admissible system behaviors. Incremental elaboration and reasoning through composition of partial models is then made possible. The price to pay is the need for handling the frame problem. In our case, we introduce two built-in axioms within our semantics to relieve the specifier from explicitly stating everything that does not change.

*Frame axiom 1: Any state variable not declared in the output clause of the specification of an operation is left unchanged by any application of this operation.*

This frame axiom is enforced by requiring the *DomPost* and *ReqPost* conditions of an operation to refer only to those state variables which are explicitly declared in the output clause of the operation (in a way similar to LARCH).

*Frame axiom 2: Every state transition that satisfies the domain pre- and postconditions of an operation corresponds to an application of this operation:*

for any operation *op*:
DomPre (op) $\wedge\circ$ DomPost (op) $\Rightarrow$ Performed (op)

## 5. Checking goal refinements

A first kind of RE-specific model verification consists in checking that the refinements of non-soft goals in the goal model are correct and complete. Such checking is important as missing subgoals result in incomplete requirements.

We first need a more precise definition of what it means for a goal refinement to be correct.
A set of goals {G1, ..., Gn} correctly refines a goal *G* in a domain theory *Dom* iff

| | |
|---|---|
| $\{G_1, \ldots, G_n, Dom\} \models G$ | *completeness* |
| $\{G_1, \ldots, G_n, Dom\} \not\models$ **false** | *consistency* |
| $\{\bigwedge_{j \neq i} G_j, Dom\} \not\models G$ for each $i \in [1..n]$ | *minimality* |

Several approaches can be followed to verify the correctness of a goal refinement.

***Approach 1: Theorem proving.*** We might use a temporal logic theorem prover - such as STeP, for example [Man96]. This is obviously a heavyweight approach requiring the assistance of an expert user. Moreover we get no real clue in case the verification fails.

***Approach 2: Formal refinement patterns.*** A more lightweight and constructive approach consists in using formal patterns to check, complete, or explore refinements ([Dar96], [Let02a]). The idea is to build a catalogue of common refinement patterns that encode refinement tactics. The patterns in the catalogue are proved formally correct once for all, e.g., using the STeP theorem prover. They are then reused in matching situations through instantiation of their meta-variables. Fig. 7 shows two frequent refinement pat-

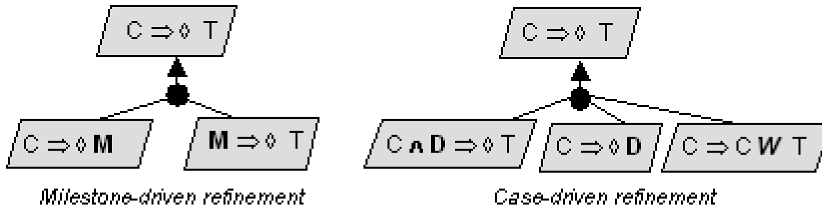



**Figure 7.** Formal refinement patterns

terns. The first pattern encodes the tactics of introducing an intermediate milestone goal whereas the second pattern encodes a standard case analysis pattern.

Fig. 8 illustrates the use of the case-driven pattern from Fig.7 in a situation where a refinement of the parent goal *Achieve[TrainProgress]* into the two left subgoals *Achieve[ProgressWhenGo]* and *Achieve[SignalSetToGo]* is being checked. An incomplete refinement is detected by pattern matching. The match reveals the missing subgoal, indicated by a dashed line in the instantiated refinement, namely, that the train must be waiting on its current block until it moves to the next block.

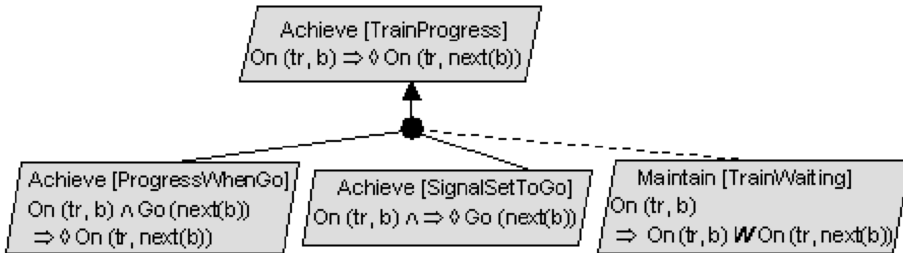



**Figure 8.** Pointing out missing subgoal through pattern instantiation [Lam07]

Refinement patterns support a constructive approach to refinement correctness. When a goal is being partially refined, we can retrieve all matching patterns from the catalogue and thereby explore alternative ways of completing the partial refinement [Dar96]. Fig. 9 illustrates that point. Three alternative subgoals appear as possible response to the refinement query on the left-hand side. Once instantiated the three returned alternatives should be assessed with respect to the application's non-functional goals to select the one that meets them best ([Chu00], [Let04]).

Another benefit of refinement patterns is the formal correctness proof of the instantiated refinement that we get for free. Each pattern in the catalogue is proved once

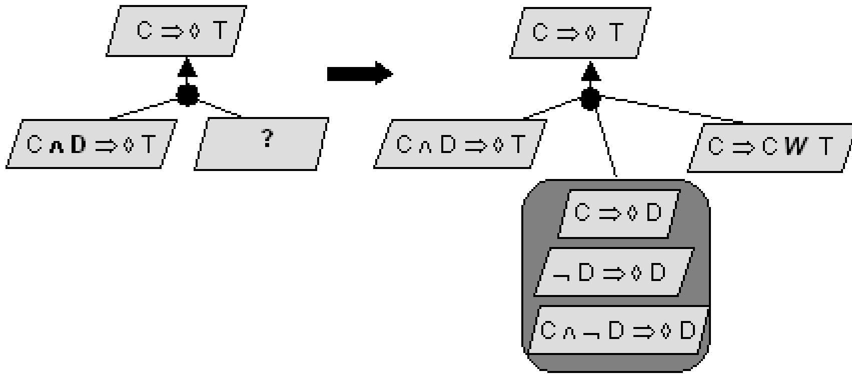



**Figure 9.** Generating alternative refinements [Lam07]

for all. For example, the proof of the case-driven pattern in Fig. 7 looks like this:

| | |
|---|---|
| 1. $C \Rightarrow \Diamond D$ | Hyp |
| 2. $C \wedge D \Rightarrow \Diamond T$ | Hyp |
| 3. $C \Rightarrow C\ \boldsymbol{W}\ T$ | Hyp |
| 4. $C \Rightarrow (C\ \boldsymbol{U}\ T) \vee \Box C$ | 3, def of Unless |
| 5. $C \Rightarrow \Diamond T \vee \Box C$ | 4, def of Until |
| 6. $C \Rightarrow \Diamond D \wedge ( \Diamond T \vee \Box C)$ | 1, 5, strengthen consequent |
| 7. $C \Rightarrow ( \Diamond D \wedge \Diamond T) \vee ( \Diamond D \wedge \Box C)$ | 6, distribution |
| 8. $C \Rightarrow ( \Diamond D \wedge \Diamond T) \vee \Diamond(D \wedge C)$ | 7, trivial lemma |
| 9. $C \Rightarrow ( \Diamond D \wedge \Diamond T) \vee \Diamond\Diamond T$ | 8, 2, strengthen consequent |
| 10. $C \Rightarrow ( \Diamond D \wedge \Diamond T) \vee \Diamond T$ | 9,$\Diamond$-idempotence |
| 11. $C \Rightarrow \Diamond T$ | 10, absorption |

Instead of having to redo such tedious proofs at every goal refinement when we build the goal model for the application, we get a proof when using a pattern just by instantiating the generic proof accordingly.

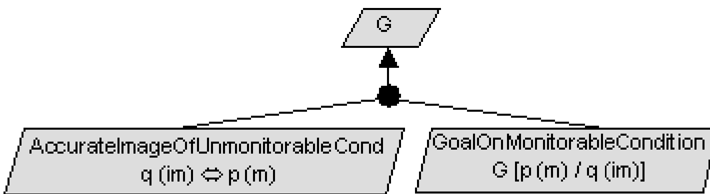



**Figure 10.** "Introduce accuracy subgoal" pattern

Some of the refinements in the pattern catalogue might be seen as high-level RT-LTL inference rules. Others are specifically aimed at refining goals towards subgoals that are realizable as defined in Section 2.3. They introduce finer-grained subgoals to resolve unrealizability problems [Let02a]. Fig. 10 shows one such pattern. The root goal *G* there

involves a condition *p(m)* on a variable *m* unmonitorable by the agent candidate for responsibility assignment. To resolve this unmonitorability, a monitorable "image" variable *im* and condition *q(im)* are introduced under the constraint that they must accurately reflect their unmonitorable counterpart.

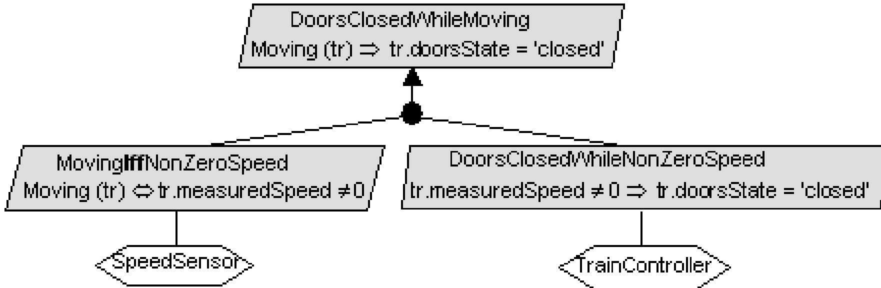



**Figure 11.** Using the "Introduce accuracy subgoal" pattern

Fig. 11 illustrates the use of this pattern on our running example. This example also suggests how such patterns are helpful for producing agent assignments as well.

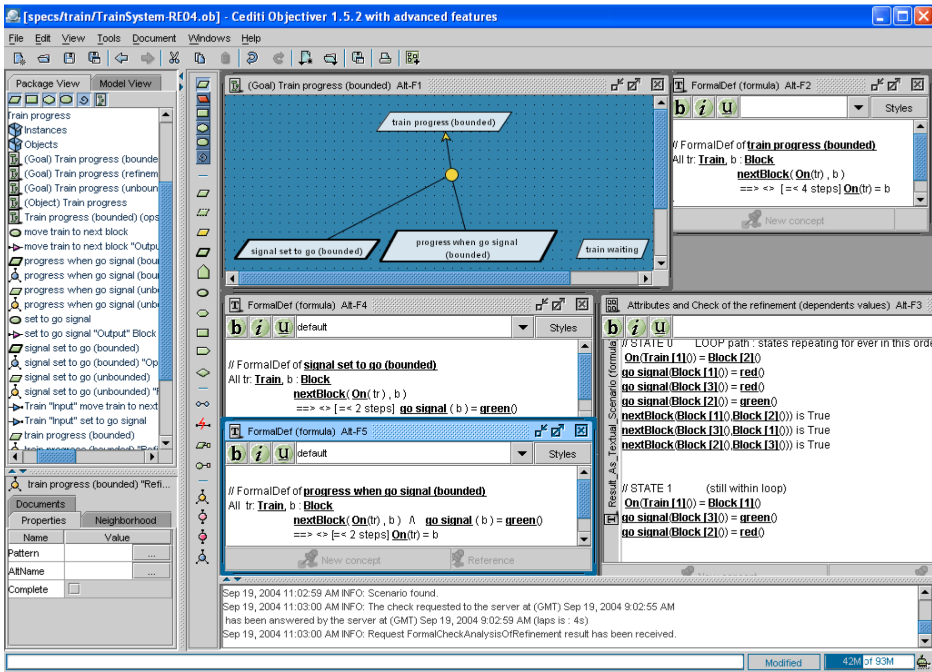



**Figure 12.** Roundtrip use of a bounded SAT solver for checking goal refinements

***Approach 3: Bounded SAT solver.*** Beside using a theorem prover or a catalogue of formal refinement patterns, we can also make a roundtrip use of a bounded SAT solver. In view of the above definition of correct refinement of goal *G* into subgoals $G_1, \ldots, G_n$, we would like to know whether the temporal logic formula

$$G_1 \land \ldots \land G_n \land Dom \land \neg G$$

is satisfiable and, if so, find a historical sequence of states satisfying it.

To achieve this, we can build a front-end that (a) asks the user to instantiate the above formula to selected object instances, in order to obtain a propositional formula, (b) translates the result into the input format required by the SAT solver, (c) asks the user to determine a maximal length to bound counterexample traces, (d) runs the SAT solver, and (e) translates the output back to the level of abstraction of the input model.

Fig. 12 shows the result produced by the FAUST tool [Pon04] on the incomplete refinement suggested in Fig. 8. The counterexample generated on the right lower window is a scenario showing the train getting back to the previous block thereby suggesting the missing subgoal of the train waiting on its current block until the signal is set to "go".

Such use of a bounded SAT solver allows partial goal models to be checked and debugged incrementally as the model is being built. The major payoff resides in the counterexample traces that may suggest missing subgoals. A bounded universe, however, makes it possible to show the presence of bugs in a goal model, not their absence.

## 6. Deriving goal operationalizations

Another kind of RE-specific model verification consists in checking the correctness of operationalizations of goals from the goal model into specifications of operations from the operation model. Such checking is important too; we must make sure that the operational specifications meet the intentional ones.

To perform such checks formally we first need a temporal logic semantics for operations [Let02b]. Such semantics is easily provided from the definition of pre-, post-, and trigger conditions in Section 2.4 and the semantic considerations in Section 4. Let *op* denote an operation from the operation model, and let

$$[|\ \text{op(in, out)}|] =_{def} \text{DomPre}\ (op) \land \circ\, \text{DomPost}(op)$$

The semantics of required pre-, trigger-, and postconditions is then:

**If** *R* $\in$ ReqPre(*op*) **then**

$$[|\ R\ |] =_{def} (\forall \star)\ ([|\ op\ |] \Rightarrow R)$$

**If** *R* $\in$ ReqTrig (*op*) **then**

$$[|\ R\ |] =_{def} (\forall \star)\ (R \land \text{DomPre}(op) \Rightarrow [|\ op\ |])$$

**If** *R* $\in$ ReqPost(*op*) **then**

$$[|\ R\ |] =_{def} (\forall \star)\ ([|\ op\ |] \Rightarrow\ \circ R)$$

Next we need a more precise definition of what it means for a goal to be correctly operationalized into operational specifications.

A set of required conditions $R_1, \ldots, R_n$ on operations from the operation model correctly operationalizes a goal *G* *iff*

$[|\ R_1\ |] \land \ldots \land [|\ R_n\ |] \models \text{G}$ *completeness*

$[|\ R_1\ |] \land \ldots \land [|\ R_n\ |] \not\models \textbf{false}$ *consistency*

$\text{G} \models [|\ R_1\ |] \land \ldots \land [|\ R_n\ |]$ *minimality*

Every operationalization defines a proof obligation. Several approaches can be followed to verify the correctness of a goal operationalization.

***Approach 1: Bounded SAT solver.*** Like for checking goal refinements, we can make a roundtrip use of a bounded SAT solver. We would now like to know whether the temporal logic formula

$$[|\ R_1\ |] \wedge \ldots \wedge [|\ R_n\ |] \wedge \text{Dom} \wedge \neg\ \text{G}$$

is satisfiable and, if so, find a historical sequence of states satisfying it. The FAUST toolset proceeds similarly to check bounded operationalizations and generate counterexample traces [Pon04].

***Approach 2: Formal operationalization patterns*** [Let02b]. The principle is similar to goal refinement patterns. A catalogue of operationalization patterns is built and formally proved correct (e.g., using the STeP theorem prover). The patterns cover common goal specification patterns [Dwy99], e.g., *Achieve* goals of form $C \Rightarrow \diamond_{\leq d}T$ or $C \Rightarrow \circ T$, and *Maintain* goals of form $C \Rightarrow T$, $C \Rightarrow \square T$, $C \Rightarrow T\ W\ N$, or $T \Rightarrow \bullet C$. The patterns are then reused in matching situations through instantiation of their meta-variables. Fig. 13 shows a pattern for operationalizing *Immediate Achieve* goals. If we apply it to the following safety goal on train signals:

∀ b: Block
[(∃ tr: Train) On (tr, b)] ⇒ ∘ ¬ GO (b)

we obtain two operations, SetSignalToStop(b) and SetSignalToGo(b), say, with trigger condition *(∃ tr: Train) On (tr, b)* on the operation SetSignalToStop(b) and a required precondition ¬ *(∃ tr: Train) On (tr, b)* on the operation SetSignalToGo(b).

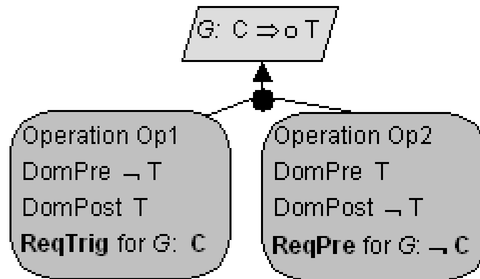



**Figure 13.** Operationalization pattern for *Immediate Achieve* goals

## 7. Obstacle analysis for mission-critical systems

Obstacle models were introduced in Section 2.5 as a means for anticipating what could go wrong in an overideal system. Goal completeness is increased through countermeasures to obstacles. This section overviews how obstacle analysis can be made further formal, in particular, through a calculus for generating obstacles from goals [Lam00b].

An overall procedure for obstacle analysis looks like this:

for every leaf goal in the goal refinement graph (requirement or expectation):

(a) identify as many obstacles to it as possible;

(b) assess their feasibility, likelihood, and severity;

(c) resolve the feasible ones according to their likelihood and severity.

Our focus here will be on steps (a) and (c). We discuss them successively.

### *7.1. Generating obstacles abductively from goal specifications*

For a goal *G*, we are looking for feasible conditions *O* such that:

$\{O, Dom\} \vdash \neg G$

$Dom \nvdash \neg O$

(see Section 2.5). We may proceed as follows :

- negate *G*;
- find as many AND/OR refinements of ¬ *G* as possible in view of properties in *Dom*,
- until obstruction preconditions are reached that are satisfiable by the environment of the set of agents assigned to *G*.

This amounts to constructing a *goal-anchored* fault-tree [Lev95] in a systematic way. To proceed more formally we need more precise definitions first.

A set of obstacles O1, ..., On is a *correct refinement* of some obstacle O in a domain theory Dom iff

| | |
|---|---|
| $\{O_1, \ldots, O_n, \text{Dom}\} \models \text{O}$ | refinement completeness |
| $\{O_1, \ldots, O_n, \text{Dom}\} \not\models \textbf{false}$ | domain consistency |
| $\{\bigwedge_{j \neq i} O_j, \text{Dom}\} \not\models \text{O}$ for each $i \in [1..n]$ | minimality |

A set of obstacles $O_1, \ldots, O_n$ to some goal *G* is *domain-complete* iff

$\{\neg O_1, \ldots, \neg O_n, \text{Dom}\} \models \text{G}$

Note that the notion of obstacle completeness is relative to what we know about the domain.

The obstacle trees we want to build should produce correct refinements of the goal negation; the leaf goals must be satisfiable by the environment of the set of agents assigned to the goal and should, ideally, form a domain-complete set of obstacles.

To generate such obstacles abductively from the goal negation and the set of known domain properties, we may follow two approaches [Lam00b].

***Approach 1 : Regression of the goal negation through the domain theory.*** This amounts to calculating preconditions for deriving $\neg G$ from *Dom*. Assuming domain properties to take the general form $A \Rightarrow C$, the procedure is as follows:

Initial step:

  take O := ¬ G

Inductive step:

  let A ⇒ C be the domain property selected,

    with *C* matching some *L* in *O* whose occurrences are all positive in *O* [Man92]

  then $\mu$ := mgu (*L*, *C*) (mgu: most general unifier)

    O := O [L / A . $\mu$ ]

Every iteration of the inductive step produces finer sub-obstacles. This technique is a counterpart, for declarative statements, of Dijkstra's precondition calculus [Dij76]. A variant of it has been used for long in AI planning [Wal77].

Let us illustrate this calculus on the generation of a well-known obstacle that caused a major accident during aircraft landing at Warsaw airport [Lad95].

We provide some context first. One goal for control of landing states (in simplified form):

MovingOnRunway $\Rightarrow$ $\circ$ ReverseThrustEnabled

As the autopilot software cannot monitor the variable MovingOnRunway, we apply the "*Introduce Accuracy Subgoal*" refinement pattern in Fig. 10 to produce the following subgoals:

MovingOnRunway $\Leftrightarrow$ WheelsState = 'turning'

WheelsState = 'turning' $\Rightarrow$ $\circ$ ReverseThrustEnabled

The second subgoal is a requirement on the autopilot software. The first subgoal is refined in the following assertions:

MovingOnRunway $\Leftrightarrow$ WheelsTurning

WheelsTurning $\Leftrightarrow$ WheelsState = 'turning'

The second assertion is an expectation on the wheel sensor. The first assertion states two assumptions made about the domain. The first says:

MovingOnRunway $\Rightarrow$ WheelsTurning

Let us try to break this assumption for obstacle analysis. We start by negating it:

$\Diamond$ MovingOnRunway $\wedge$ $\neg$ WheelsTurning

We refine this negation by regressing it through the domain. We look for, or elicit, domain properties that are *necessary conditions* for the target condition *WheelsTurning* in the assumption we want to obstruct. We might find, in particular,

WheelsTurning $\Rightarrow$ WheelsOut,

WheelsTurning $\Rightarrow$ $\neg$ WheelsBroken,

WheelsTurning $\Rightarrow$ $\neg$ Aquaplaning, etc.

Let us select the third domain property, equivalent to its contraposition:

Aquaplaning $\Rightarrow$ $\neg$ WheelsTurning

The consequent of this implication unifies with one of the conjuncts in the negated assumption above. We may therefore regress that negated assumption backwards through this domain property which yields the following subobstacle obstructing the target assumption:

$\Diamond$ MovingOnRunway $\wedge$ Aquaplaning

The generated obstacle is satisfiable by the environment (in this case, mother Nature). It was indeed satisfied during the Warsaw crash.

As the above derivation suggests, obstacle analysis may be used to elicit unknown domain properties as well.

***Approach 2 : Formal obstruction patterns.*** We can again build a catalogue of common goal obstruction patterns, prove each of them, and reuse them by instantiation in matching situations [Lam00b]. Fig. 14 shows a very common obstruction pattern. The pattern encodes a single regression step. The above derivation can thus be seen as an application of this pattern as well.

Once generated, the obstacles need to be assessed for feasibility, likelihood and severity. For feasibility, a SAT solver might be used to check that the obstacle assertion is satisfiable by the environment. For likelihood and severity, standard risk management techniques should be used.

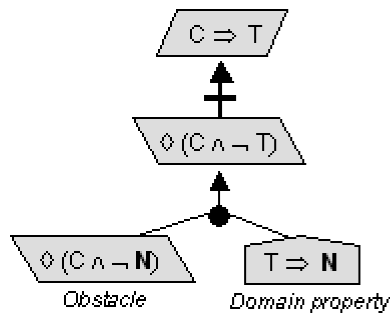



**Figure 14.** Goal obstruction pattern

## *7.2. Resolving obstacles*

The generated obstacles, if feasible and likely, must be resolved through countermeasures. Resolution may be undertaken at requirements engineering time or deferred to system runtime through obstacle monitoring [Fea98]. For resolution at RE time, we may (a) explore alternative resolutions by application of model transformation operators [Lam00b], and then (b) select a "best" resolution based on the likelihood and severity of the obstacle, and on other non-functional goals from the goal model [Chu00].

Model transformation operators encode various resolution tactics such as the following.

- *Goal substitution*: Consider alternative refinements of the parent goal to avoid the obstruction of a child goal - e.g., replace the obstructed subgoal MotorReversedIffWheelsTurning by the goal MotorReversedIffPlaneWeightSensed.
- *Agent substitution*: Consider alternative responsibility assignments for the obstructed goal - e.g., replace the agent OnBoardTrainController, assigned to some obstructed safety-critical goal in the train control system, by the agent VitalStationComputer.
- *Goal weakening*: Weaken the goal formulation - e.g., weaken the goal SectorTrafficControllerOnDuty in an air traffic control system into the goal SectorTrafficControllerOnDutyOrWarningToNextSector.
- *Goal restoration*: Enforce the target condition in the obstructed goal when the obstacle occurs - e.g., generate an alarm to the pilot in case the obstacle WheelsNotOut occurs.
- *Obstacle prevention*: Introduce a new Avoid goal, to be refined in turn, in order to prevent the obstacle from occurring - e.g., introduce in the goal model a new goal Avoid[TrainAccelerationCommandCorrupted].
- *Obstacle mitigation*: Tolerate the obstacle but mitigate its effects - e.g., introduce a new goal Avoid[TrainCollisionWhenOutDatedTrainInfo].

## 8. Threat analysis for security-critical systems

As introduced in Section 2.6, threats are malicious obstacles obstructing security goals. Threat analysis consists in identifying threats to the system and resolving them through countermeasures. It involves an anti-model, that is, a dual model of threats to the system model. Such model shows how security goals can be obstructed by linking negated goals to the attacker's malicious goals, called *anti-goals*, and capabilities.

The attacker's *capabilities* are captured by two sets of conditions that the attacker can monitor and control, respectively. These capabilities define the interface between the attacker and its own environment, including the threatened software-to-be. The properties of the attacker's environment includes the properties of the software-to-be, including monitorable vulnerabilities to be exploited for anti-goal achievement.

The attacker is a system agent who knows (a) the application's goal model (b) all descriptive domain properties used to build it, and (c) the operation model. In the tradition of the *Most Powerful Attacker* model used in cryptographic protocol analysis ([Kem94], [Low96], [Cla00]), we assume a *Most Knowledgeable Attacker* (MKA) that knows everything about the application model being attacked. Worst-case analysis of threats is required to ensure the completeness of the set of countermeasures to them. The MKA assumption is trivially satisfied here as the attacker at RE time is the application modeller looking for missing countermeasures. Such MKA model also implies that the attacker has no need to dynamically increase its knowledge through observation of system behaviors in response to attacker's stimuli - he has that knowledge already.

An overall procedure for threat analysis looks like this:

1. Build threat graphs rooted on antigoals:
   (a) Get initial anti-goals as roots;
   (b) Identify classes of attackers wishing these, and their capabilities;
   (c) For each root anti-goal and attacker class:
      build an anti-goal refinement graph as a proof that the root anti-goal can be satisfied in view of the attacker's knowledge and capabilities; refinement terminates when leaf conditions are reached that meet the attacker's capabilities.
2. Derive new security goals as countermeasures to counter anti-goals from threat graphs.

We first review the types of security goals that might be threatened by an anti-model together with their specification patterns. Next we will overview and illustrate the various steps of the above procedure. Our main focus will be on formal support for steps (a) and (c).

### *8.1. Specification patterns for security goals*

Security goals prescribe different types of protection of system assets. Numerous taxonomies of security properties are available from the literature - see, e.g., [Kem03]. We can formally specify property classes to define corresponding specification patterns on meta-variables. Instantiating these meta-variables to application-specific, sensitive objects provides candidate security goals for our system, to be refined in the goal model and to be obstructed in an anti-goal model [Lam04a].

For example, the specification pattern for *confidentiality* goals defines confidentiality in a generic way as follows:

**Goal** *Avoid* [SensitiveInfoKnownByUnauthorizedAgent]
**FormalSpec** $\forall$ ag: Agent, ob: Object
$\neg$ Authorized(ag, ob.Info) $\Rightarrow$ $\neg$ KnowsV$_{ag}$ (ob.Info)

To specify security goals, our real-time linear temporal logic is augmented with epistemic constructs [Fag95]. In particular, the operator KnowsV$_{ag}$ is defined on state variables as follows:

KnowsV$_{ag}$(v) $\equiv$ $\exists$ x : Knows$_{ag}$(x=v) ("knows value")
Knows$_{ag}$(P) $\equiv$ Belief$_{ag}$(P) $\wedge$ P ("knows property")

The operational semantics of the epistemic operator Belief$_{ag}(P)$ is: "*P is among the properties stored in the local memory of agent* ag". Domain-specific axioms must make it precise under which conditions property *P* does appear and disappear in the agent's memory. An agent thus *knows a property* if that property is found in its local memory and it is indeed the case that the property holds.

In the above pattern for confidentiality goals, the Authorized predicate is a generic predicate to be instantiated through a domain-specific definition. For example, for web banking services we would certainly consider the instantiation Object/Account while searching through the object model for sensitive information to be protected. We might then introduce the following instantiating definition:

$\forall$ ag: Agent, acc: Account
Authorized (ag, acc) $\equiv$ Owner (ag, acc) $\vee$ Proxy (ag, acc) $\vee$ Manager (ag, acc)

Sensitive information about accounts includes the objects Acc# and PIN. The latter are defined in the object model as entities composing the aggregated entity Account and linked through a *Matching* association.

Instantiating the *Confidentiality* specification pattern to this sensitive information yields the following confidentiality goal as candidate for inclusion in the goal model for web banking services:

**Goal** *Avoid* [AccountNumber&PinKnownByUnauthorized]
**FormalSpec** $\forall$ p: Person, acc: Account
$\neg$ (Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc) )
$\Rightarrow$ $\neg$ [KnowsV$_p$ (acc.Acc#) $\wedge$ KnowsV$_p$ (acc.PIN)]

Other patterns may be defined for specifying and eliciting application-specific instantiations of privacy, integrity, availability, authentication, accountability, or non-repudiation goals, e.g.,

**Goal** *Maintain* [PrivateInfoKnownOnlyIfAuthorizedByOwner]
**FormalSpec** $\forall$ ag, ag': Agent, ob: Object
KnowsV$_{ag}$ (ob.Info) $\wedge$ OwnedBy (ob.Info, ag') $\wedge$ ag $\neq$ ag'
$\Rightarrow$ AuthorizedBy (ag, ob.Info, ag')

**Goal** *Maintain* [ObjectInfoChangeOnlyIfCorrectAndAuthorized]
**FormalSpec** $\forall$ ag: Agent, ob: Object, v : Value
ob.Info = v $\wedge \circ$ (ob.Info $\neq$ v) $\wedge$ UnderControl (ob.Info, ag)
$\Rightarrow$ Authorized (ag, ob.Info) $\wedge \circ$ Integrity (ob.Info)

**Goal** *Achieve* [ObjectInfoUsableWhenNeededAndAuthorized]
**FormalSpec** $\forall$ ag: Agent, ob: Object, v : Value
[Needs (ag, ob.Info) $\wedge$ Authorized (ag, ob.Info)] $\Rightarrow$ $\Diamond_{\leq d}$ Using (ag, ob.Info)

Specifications of application-specific security goals are thus obtained from such patterns by (a) instantiating meta-classes such as Object, Agent, and generic attributes such as *Info*, to application-specific sensitive classes, attributes and associations in the object model; and (b) specializing predicates such as Authorized, UnderControl, Integrity, or Using through substitution by application-specific definitions.

The specification patterns can be diversified through variants capturing different security options. For confidentiality goals, for example, we may consider variants along two dimensions: (a) the degree of approximate knowledge to be kept confidential - exact value of a state variable, or lower/upper bound, or order of magnitude, or any property about the value; and (b) the timing according to which that knowledge should be kept confidential - confidential now, or confidential until some expiration date, or confidential unless/until condition, or confidential forever [Del05].

### 8.2. Identifying initial anti-goals and attackers

Preliminary anti-goals must be identified as root threats to be refined in threat graphs. One obvious option is to browse the goal model systematically in order to determine whether there are any goal negations that could be wished by malicious agents.

For example, while browsing the goal model for an online shopping system we might stop on the goal stating that every purchased item must have been paid within two days before being sent:

ItemSentToBuyer $\Rightarrow$ $\blacklozenge_{\leq 2d}$ ItemPaidToSeller

(The goal is specified propositionally for simplicity.) The goal negation is:

$\Diamond$ (ItemSentToBuyer $\wedge \neg \blacklozenge_{\leq 2d}$ ItemPaidToSeller)

This goal is obviously going to be wished by a number of malicious shoppers. We should therefore consider it among the root anti-goals for threat graph building.

We can also directly obtain root anti-goals by negating security goal patterns instantiated to application-specific sensitive objects. For example, the negation of the instantiated confidentiality goal *Avoid* [AccountNumber&PinKnownByUnauthorized] for web banking services yields another initial anti-goal:

**AntiGoal** *Achieve* [AccountNumber&PinKnownByUnauthorized]
**FormalSpec** $\Diamond \exists$ p: Person, acc: Account
$\neg$ [ Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc) ]
$\wedge$ KnowsV$_p$ (acc.Acc#) $\wedge$ KnowsV$_p$ (acc.PIN)

The identification of attacker classes is obviously intertwined with the identification of initial anti-goals; the negation of an application-specific goal raises the question of who might benefit from it. We may also use attacker taxonomies available from the literature to identify attackers.

For example, by asking who could benefit from the anti-goal *Achieve* [AccountNumber&PinKnownByUnauthorized] we could elicit agent classes such as Thief, Hacker, BankQualityAssuranceTeam, etc.

### *8.3. Building threat graphs*

For each initial anti-goal and attacker class identified, we need to build an anti-goal refinement/abstraction graph as a basis for exploring countermeasures.

We can do this informally, like for any goal model, by asking *WHY* questions to identify parent anti-goals, and *HOW* questions to identify child anti-goals.

When the goals, domain properties, and anti-goals are specified formally we can use the regression technique presented in Section 7. The difference now is that the anti-goal regression should be applied not only to domain properties but also to requirements and expectations as the attacker should exploit these as well. We will thereby obtain anti-goal preconditions to be satisfied by the attacked software and its environment.

Whatever technique is used, the anti-goal refinement along a branch stops as soon as we obtain a precondition which is monitorable or controllable according to the attacker's capabilities.

Let us illustrate the construction of a threat graph for web banking services using a mix of informal and formal techniques.

We take a Thief agent, for example. Starting from the above anti-goal *Achieve* [AccountNumber&PinKnownByUnauthorized], we obtain through *WHY* questions a parent anti-goal *Achieve*[PaymentMediumKnownByThief] and a grand-parent anti-goal *Achieve* [MoneyStolenFromBankAccounts], see Fig. 15. The milestone refinement pattern in Fig. 7 produces two other subgoals of the parent anti-goal *Achieve* [PaymentMediumKnownByThief], namely, *Achieve* [ThiefKnowsWhichBank] and *Achieve* [ThiefKnowsAccountStructure].

Let us focus on the derivation of refinements for the anti-goal *Achieve* [AccountNumber&PinKnownByUnauthorized]. Looking at the formal specification of this anti-goal, obtained earlier as negation of an instantiated confidentiality goal pattern, we ask ourselves "*what are sufficient conditions in the domain for someone unauthorized to know both the number and PIN of an account simultaneously?*". We may also use the symmetry of the association Matching between account numbers and PINs in the object model and its multiplicity [1..1, 1..N]. As a result we find in *Dom*, or elicit, two symmetrical domain properties:

∀ p: Person, acc: Account
¬ [Owner (p, acc) ∨ Proxy (p, acc) ∨ Manager (p, acc)] ∧ KnowsV$_p$ (acc.Acc#)
  ∧ ( ∃ x: PIN) (Found (p, x) ∧ Matching (x, acc.Acc#))
  ⇒ KnowsV$_p$ (acc.Acc#) ∧ KnowsV$_p$ (acc.PIN)
¬ [Owner (p, acc) ∨ Proxy (p, acc) ∨ Manager (p, acc) ] ∧ KnowsV$_p$ (acc.PIN)
  ∧ ( ∃ y: Acc#) (Found (p, y) ∧ Matching (acc.PIN, y))
  ⇒ KnowsV$_p$ (acc.Acc#) ∧ KnowsV$_p$ (acc.PIN)

Now we may regress the anti-goal *Achieve*[AccountNumber&PinKnownByUnauthorized] through each of these domain properties to obtain two sub-goals as alternative preconditions for achieving this anti-goal. We thereby obtain an OR-refinement of that anti-goal into two alternative, symmetrical anti-subgoals, namely,

**AntiGoal** *Achieve* [AccountKnown&MatchingPinFound]
  **FormalSpec** ◇ ∃ p: Person, acc: Account
    ¬ [Owner (p, acc) ∨ Proxy (p, acc) ∨ Manager (p, acc) ]
    ∧ KnowsV$_p$ (acc.Acc#)
    ∧ ( ∃ x: PIN) [ Found (p, x) ∧ Matching (x, acc.Acc#) ]

**AntiGoal** *Achieve* [PinKnown&MatchingAccountFound]
**FormalSpec** $\Diamond \exists$ p: Person, acc: Account
$\neg$ [Owner (p, acc) $\vee$ Proxy (p, acc) $\vee$ Manager (p, acc) ]
$\wedge$ KnowsV$_p$ (acc.PIN)
$\wedge$ ( $\exists$ y: Acc#) [ Found (p, y) $\wedge$ Matching (acc.PIN, y) ]

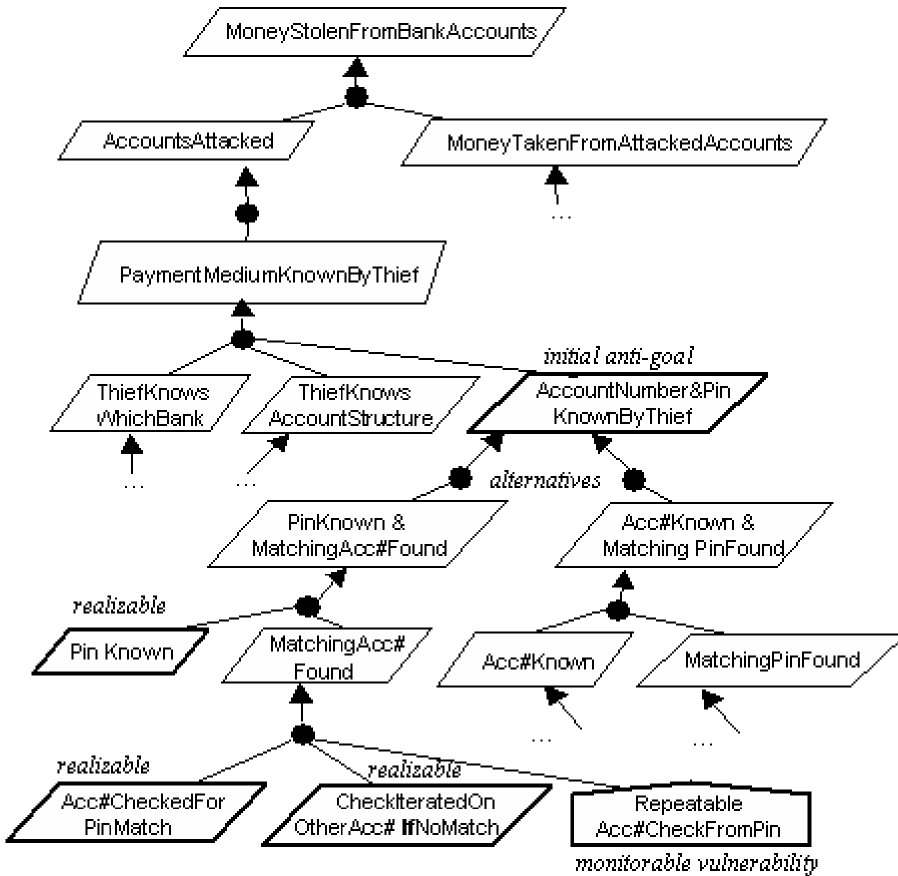



**Figure 15.** Threat graph fragment for web banking services [Lam04a]

The refinement process goes on until reaching terminal conditions that are either realizable anti-requirements in view of the attacker's capabilities or observable vulnerabilities of the attacker's environment (including the target software). Fig. 15 shows the threat graph obtained.

The derived anti-requirements on the Thief agent are, in the alternative refinement shown in Fig. 15,

AccountNumberCheckedForPinMatch,
CheckIteratedOnOtherAccountNumbers**If**NoMatch

These anti-requirements are realizable under the anti-domain property RepeatableAcc#-

CheckFromPin, stating that the attacker can iterate on account numbers to check whether they match some fixed 4-digit number.

The threat graph in Fig. 15 with this alternative branch corresponds to a real attack reported in [Dos00]. Along the other alternative, not fully elaborated in Fig. 15, we reach symmetrical leaf goals

PinCheckedForAccountNumberMatch,
CheckIteratedOnOtherPins**If**NoMatch,
RepeatablePinCheckFrom Acc#

The two first subgoals are realizable anti-requirements whereas the third condition is a vulnerability precluded by banking systems. This alternative is thus not realizable.

Recent efforts have been devoted to synthesizing threat graphs fully automatically and efficiently [Jan06]. Based on a BDD representation of the initial anti-goal, the technique consists in generating a proof showing that this anti-goal is realizable in view of the attacker's knowledge and capabilities. The proof amounts to a hierarchical plan for satisfying the anti-goal. The hierarchical levels in this plan are determined systematically by incrementally weakening powerful virtual macro-agents until the capabilities of the real attacker agent are reached. The weakening consists in removing macro-agent capabilities by following the anti-goal's BDD state-variable ordering.

### *8.4. Deriving countermeasures*

Based on the threat graphs built for each initial anti-goal, we may obtain new security goals by application of the resolution operators reviewed in Section 7.

In security-critical systems the operator *Avoid[X]* is frequently used, where *X* is instantiated to an anti-goal or a vulnerability. For example, in our web banking system, we would certainly take the following goals as new goals to be refined:

Avoid [RepeatableAcc#CheckFromPin]
Avoid [RepeatablePinCheckFrom Acc#]

Resolution operators can be further specialized to malicious obstacles; in particular, the two following tactics should be considered for countermeasures:

- Make vulnerability condition unmonitorable by attackers.
- Make anti-requirement uncontrollable by attackers.

The alternative countermeasures obtained through such resolution operators must be refined in turn along alternative OR-branches of the updated goal model. Such alternatives must be assessed to keep some "best" one in view of the other non-functional goals [Chu00] and the conflicts they often introduce with other goals in the goal model (see next section). A new threat analysis cycle may need to be undertaken for these new goals.

The threat analysis method reported in this section was applied in the European SAFEE project to model and analyze on-board terrorist threats against civil aircrafts, and explore corresponding countermeasures. A goal model with derived countermeasures was used as a basis for elaborating the requirements for an on-board threat detection/reaction system.

## 9. Conflict analysis

Requirements engineers are faced with numerous conflicts while elaborating system goals, requirements, and expectations. Conflicts arise from multiple viewpoints among different stakeholders [Fin94], or from different categories of functional and non-functional goals that are potentially conflicting - for example, safety goals tend to be conflicting with performance goals. Security goal categories are especially involved in potential conflicts. For example, "maintain agent anonymity" is potentially conflicting with "achieve agent accountability"; "password-based authentication" is potentially conflicting with "application usability"; "encrypted transaction" is potentially conflicting with "efficient transaction"; and so forth.

Managing interactions among goals, requirements, and expectations is a core business in the RE process [Rob03]. Such interactions much more often amount to *potential* conflicts, rather than logical inconsistencies where one stakeholder says "I want P" whereas another says "I want $\neg P$". The notion of potential conflict is captured through the following definition.

Goals $G_1, \ldots, G_n$ are *divergent* within a domain *Dom* iff there exists a *boundary condition B* such that the following conditions hold:

| | | |
|---|---|---|
| 1. | {Dom, *B*, $\bigwedge_{1 \leq i \leq n} G_i\}$ $\models$ **false** | *potential conflict* |
| 2. | For each *i*: {Dom, *B*, $\bigwedge_{j \neq i} G_j\}$ $\not\models$ **false** | *minimality* |
| 3. | There exists a behavior *E* of the environment of the set of agents in charge of $G_1, \ldots, G_n$ such that $E \models$ O | *feasibility* |

The boundary condition captures a particular combination of circumstances which makes the goals $G_1, \ldots, G_n$ conflicting if conjoined to it (see conditions (1) and (2)). Note that a conflict is a particular case of divergence in which *B* = **true**. Also note that the minimality condition precludes the trivial boundary condition *B* = **false**; it stipulates in particular that the boundary condition must be consistent with the domain theory *Dom*. The boundary condition must also be satisfiable by the environment of the agents involved in the satisfaction of the divergent goals.

Conflict management consists in detecting conflicts among goals, generating alternative resolutions of the detected conflicts, and selecting a best resolution ([Lam98a], [Rob03]). We briefly review these steps successively for the more general notion of divergence.

### *9.1. Detecting divergences*

Similarly to obstacles, we may detect divergences among goals by regression or by use of conflict patterns [Lam98a].

***Approach 1: Regression.*** The technique is based on the observation that the first condition for divergence is equivalent to:

$$\{\text{Dom}, B, \bigwedge_{j \neq i} G_j \} \models \neg G_i$$

We may thus formally derive the boundary condition *B* as precondition for one of the negated goals $\neg G_i$, chaining backwards through an augmented theory $\{\text{Dom}, \bigwedge_{j \neq i} G_j\}$. The regession procedure is similar to the one given in Section 7.

Let us illustrate how a divergence can thereby be detected between two typical security goals, taken from a real situation [Lam98a]. Consider the electronic reviewing process for a scientific journal, with the following two security goals:

**Goal** *Maintain* [ReviewerAnonymity]
  **FormalSpec** $\forall$ r: Reviewer, p: Paper, a: Author, rep: Report
    Reviews (r, p, rep) $\wedge$ AuthorOf (a, p) $\Rightarrow\ \Box\neg$ KnowsV$_a$ (Reviews[r,p,rep])

**Goal** *Maintain* [ReviewIntegrity]
  **FormalSpec** $\forall$ r: Reviewer, p: Paper, a: Author, rep, rep': Report
    AuthorOf (a, p) $\wedge$ Gets (a, rep, p, r) $\Rightarrow$ Reviews (r, p, rep') $\wedge$ rep' = rep

In this specification, the object Reviews[r,p,rep] designates a ternary association capturing a reviewer *r* having produced a referee report *rep* for paper *p*. The predicate *Reviews(r,p,rep)* expresses that an instance of this association exists in the current state. The predicate *Gets(a,rep,p,r)* expresses that author *a* has the report *rep* by reviewer *r* for his paper *p*. The *KnowsV* predicate is the epistemic construct introduced in Section 8.

The above goals are *not* logically inconsistent. However, let us see whether they are potentially conflicting. We take the goal *Maintain[ReviewerAnonymity]* for the initialization step of the regression procedure. Its negation is:

$\Diamond\,\exists$ r: Reviewer, p: Paper, a: Author, rep: Report (NG)
Reviews(r,p,rep) $\wedge$ AuthorOf(a,p) $\wedge\,\Diamond$ KnowsV$_a$(Reviews[r,p,rep])

Regressing (NG) through the *ReviewIntegrity* goal, whose consequent can be simplified to *Reviews(r,p,rep)* by term rewriting, yields:

$\Diamond\,\exists$ r: Reviewer, p: Paper, a: Author, rep: Report (NG1)
AuthorOf (a,p) $\wedge$ Gets (a, rep, p, r) $\wedge\,\Diamond$ KnowsV$_a$(Reviews[r,p,rep])

Let us assume that the domain theory contains the following sufficient conditions for identifiability of reviewers (the outer universal quantifiers are left implicit for simplicity):

Gets (a, rep, p, r) $\wedge$ Identifiable (r, rep) $\Rightarrow\,\Diamond$ KnowsV$_a$(Reviews[r,p,rep]) (D1)
Reviews (r, p, rep) $\wedge$ SignedBy (rep, r) $\Rightarrow$ Identifiable (r, rep) (D2)
Reviews (r, p, rep) $\wedge$ French (r)$\wedge\neg\exists$ r'$\neq$ r: (D3)
  [Expert (r', p) $\wedge$ French (r')] $\Rightarrow$ Identifiable (r, rep)

In these property specifications, the predicate *Identifiable(r,rep)* means that the identity of reviewer r can be determined from the content of report rep. Properties (D2) and (D3) provide explicit sufficient conditions for this. The predicate *SignedBy(rep,r)* means that report rep contains the signature of reviewer r. The predicate *Expert(r,p)* means that reviewer r is a well-known expert in the domain of paper p. Property (D3) states that a French reviewer notably known as being the only French expert in the area of the paper is identifiable (as she makes typical French errors of English usage).

The third conjunct in (NG1) unifies with the consequent in (D1); the regression yields, after corresponding substitutions of variables:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report
AuthorOf (a, p) $\wedge$ Gets (a, rep, p, r) $\wedge$ Identifiable (r, rep)

The last subformula in this formula unifies with the consequent in (D3); the regression yields:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report (B)
AuthorOf (a, p) $\wedge$ Gets (a, rep, p, r) $\wedge$ Reviews (r, p, rep)
$\wedge$ French(r) $\wedge \neg \exists$ r' $\neq$ r: [Expert (r', p) $\wedge$ French (r')]

This condition is satisfiable through a report produced by a French reviewer who is the only well-known French expert in the domain of the paper, and sent unaltered to the author (as variable *rep* is the same in the *Reviews* and *Gets* predicates). We thus formally derived a boundary condition making the divergent goals *Maintain[ReviewerAnonymity]* and *Maintain[ReviewIntegrity]* logically inconsistent.

The space of derivable boundary conditions can be explored by backtracking on each applied property to select another applicable one. After having selected (D3), we could select (D2) to derive another boundary condition:

$\Diamond \exists$ r: Reviewer, p: Paper, a: Author, rep: Report (B')
AuthorOf(a, p) $\wedge$ Gets(a, rep, p, r) $\wedge$ Reviews(r, p, rep) $\wedge$ SignedBy(rep, r)

which captures the situation of an author receiving the same report as the one produced by the reviewer with signature information found in it.

***Approach 2: Formal conflict patterns.*** Alternatively we may sometimes shortcut such derivations by instantiating common patterns of divergence among goals that highlight generic boundary conditions. Fig. 16 shows one such pattern that occurs frequently in practice.

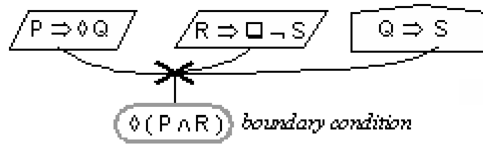



**Figure 16.** *Achieve-Avoid* divergence pattern

### 9.2. Resolving divergences

The principle here again is to generate alternative resolutions through resolution operators and then to compare them in order to select a best resolution. Here is a sample of conflict resolution operators.

- *Avoid boundary condition*: A new goal is introduced which takes the form:
  $\square \neg$ B.
- *Restore divergent goals*: A new goal is introduced which takes the form:
  $B \Rightarrow \Diamond \bigwedge_{1 \leq i \leq n} G_i$

- *Anticipate conflict*: This strategy can be applied when some persistent condition $P$ can be found such that, in some context $C$, we inevitably get into a conflict after some time if the condition $P$ has persisted over a too long period:
$$C \wedge \Box_{\leq d} P \Leftrightarrow \Diamond_{\leq d} \neg \bigwedge_{1 \leq i \leq n} G_i$$
In such a case we may introduce the following new goal to avoid the conflict by anticipation:
$$C \wedge P \Rightarrow \Diamond_{\leq d} \neg P$$
- *Weaken* the formulation of one of the divergent goals.
- *Specialize* the target objects concerned by the divergent goals so that the latter now refer to non-overlapping specializations.
- Etc. [Rob03].

In our journal reviewing example, we might resolve the detected divergence by avoiding the boundary condition (that is, not asking a French reviewer in case she is the only French expert in the domain of the paper), or by weakening a divergent goal (e.g., weakening the integrity requirement to allow for correction of typical French errors of English usage).

Conflicts should be carefully considered in safety-critical systems too. An interesting example comes from a document describing some of the requirements for the San Francisco Bay Area Rapid Transit System (BART). One goal stated that the speed commanded to trains may not be "too high", because otherwise it forces the distance between trains to be too high (for safety reasons). Another goal stated that the commanded speed may not be "too low", because otherwise it may force accelerations felt uncomfortable by passengers. These goals are more precisely specified as follows:

**Goal** *Maintain* [CmdedSpeedCloseToPhysicalSpeed]
    **FormalSpec** $\forall$ tr: Train
        $tr.Acc_{CM} \geq 0 \Rightarrow tr.Speed_{CM} \leq tr.Speed + f(distance\text{-}to\text{-}obstacle)$
**Goal** *Maintain* [CmdedSpeedAbove7mphOfPhysicalSpeed]
    **FormalSpec** $\forall$ tr: Train
        $tr.Acc_{CM} \geq 0 \Rightarrow tr.Speed_{CM} > tr.Speed + 7$

The boundary condition for making these goals logically inconsistent is easily derived:
$$\Diamond\ (\exists\ tr: Train)\ (tr.Acc_{CM} \geq 0 \wedge f(dist\text{-}to\text{-}obstacle) \leq 7)$$
The selected resolution operator should be goal weakening; we should keep the safety goal as it is and weaken the convenience goal in order to remove the divergence by covering the boundary condition:

**Goal** *Maintain* [CmdedSpeedAbove7mphOfPhysicalSpeed]
    **FormalSpec** $\forall$ tr: Train
        $tr.Acc_{CM} \geq 0 \Rightarrow tr.Speed_{CM} > tr.Speed + 7 \vee f(dist\text{-}to\text{-}obstacle) \leq 7$

## 10. Synthesizing behavior models from scenarios and goals

Goals, scenarios, and state machines form a win-win partnership for system modeling and analysis.

- Goal models support various forms of early, declarative, and incremental reasoning, as seen in the previous section. On the downside, goals are sometimes felt

too abstract by stakeholders. They cover classes of intended behaviors but such behaviors are left implicit. Goals may also be hard to elicit and make fully precise in the first place.

- Scenarios support a concrete, narrative expression style, as discussed in Section 2.7. They are easily accessible to stakeholders. On the downside, scenarios cover few behaviors of specific instances. They leave intended system properties implicit.
- State machines provide visual abstractions of explicit behaviors for any agent instance in some corresponding class (see Section 2.7). They can be composed sequentially and in parallel, and are executable for requirements validation through animation. They can be verified against declarative properties. State machines also provide a good basis for code generation. On the downside, state machines are too operational in the early stages of requirements elaboration. Their construction may be quite hard.

Those complementary strengths and limitations call for an approach integrating goal, scenario, and state machine models where portions of one model are synthesized from portions of the other models.

Recent efforts were made along this line. For example, a labelled transition system (LTS) model can be synthesized from message sequence charts (MSC) taken as positive examples of system behavior [Uch03]. MSC specifications can be translated into statecharts [Kru98]. UML state diagrams can be generated from sequence diagrams capturing positive scenarios ([Whi00], [Mak01]). Goal specifications in linear temporal logic can also be inferred inductively from MSC scenarios taken as positive or negative examples [Lam98b]. These various techniques all require additional input information beside scenarios, namely, a high-level message sequence chart showing how MSC scenarios are to be flowcharted [Uch03]; pre- and post-conditions of interactions, expressed on global state variables ([Lam98b], [Whi00]); local MSC conditions [Kru98]; or state machine traces local to some specific agent [Mak01]. Such additional input information may be hard to get from stakeholders, and may need to be refactored in non-trivial ways in case new positive or negative scenario examples are provided later in the requirements/design engineering process [Let05].

State machine models can be synthesized inductively from positive and negative scenarios without requiring such additional input information [Dam05]. Let us have a closer look at how this synthesis technique works. We start with some background first.

As introduced in Section 2.7, a positive scenario illustrates some desired system behavior. A negative scenario captures a behavior that may not occur. It is captured by a pair *(p, e)* where *p* is a positive MSC, called precondition, and *e* is a prohibited subsequent event. The meaning is that once the admissible MSC precondition has occurred, the prohibited event may not label the next interaction among the corresponding agents.

Fig. 17 shows a collection of input scenarios for state machine synthesis. The upper right scenario is a negative one. The intuitive, end-user semantics of two consecutive events along a MSC timeline is that the first is *directly* followed by the second. The actual semantics of MSCs is defined in terms of LTS and parallel composition [Uch03]. A MSC timeline defines a unique finite LTS execution that captures a corresponding agent behavior. Similarly, the semantics of an entire MSC is defined in terms of the LTS modeling the entire system. MSCs define executions of the parallel composition of each agent LTS.

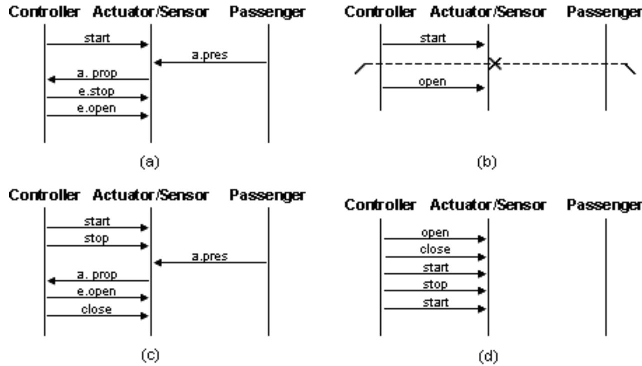



**Figure 17.** Input scenarios for a train system

For goal injection in the synthesis process, we take a fluent-based variant of LTL where the atomic assertions are explicitly defined in terms of the events making them true and false, respectively [Gia03]. A fluent Fl is a proposition defined by a set $Init_{Fl}$ of initiating events, a set $Term_{Fl}$ of terminating events, and an initial value $Initially_{Fl}$ that can be true or false. The sets of initiating and terminating events must be disjoint. A fluent definition takes the form:

fluent Fl = <$Init_{Fl}$, $Term_{Fl}$> initially $Initially_{Fl}$

In our train example, the fluents *DoorsClosed* and *Moving* are defined as follows:

fluent DoorsClosed = <{close doors}, {open doors, emergency open}> initially **true**

fluent Moving = <{start}, {stop, emergency stop}> initially **false**

A fluent *Fl* holds at some time if either of the following conditions holds:

(a) *Fl* holds initially and no terminating event has yet occurred;

(b) some initiating event has occurred and no terminating event has occurred since then.

LTS synthesis proceeds in two steps [Dam05]. First, the input scenarios are generalized into a LTS for the entire system, called *system LTS*. This LTS is then projected on each agent using standard automaton transformation algorithms [Hop79].

The system LTS covers all positive scenarios and excludes all negative ones. It is obtained by an interactive extension of a grammar induction algorithm known as RPNI [Onc92]. Grammar induction aims at learning a language from a set of positive and negative strings defined on a specific alphabet. The alphabet here is the set of event labels; the strings are provided by positive and negative scenarios.

RPNI first computes an initial LTS solution, called *Prefix Tree Acceptor* (PTA). The PTA is a deterministic LTS built from the input scenarios; each scenario is a branch in the tree that ends with a "white" state, for a positive scenario, or a "black" state, for a negative one. As in the other aforementioned synthesis approaches, scenarios are assumed to start in the same system state.

Fig. 18 shows the PTA computed from the scenarios in Fig. 17. A black state is an error state for the system. A path leading to a black state is said to be *rejected* by the LTS; a path leading to a white state is said to be *accepted* by the LTS. By construction, the PTA accepts all positive input scenarios while rejecting all negative ones.

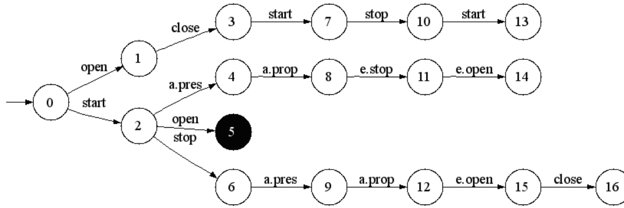



**Figure 18.** PTA built from the scenarios in Fig. 17

Behavior generalization from the PTA is achieved by a generate-and-test algorithm that performs an exhaustive search for equivalent state pairs to *merge* them into equivalence classes. Two states are considered *equivalent* if they have *no incompatible continuation*, that is, there is no subsequent event sequence accepted by one and rejected by the other.

At each generate-and-test cycle, RPNI considers merging a state *q* in the current solution with a state *q'* of lower rank. Merging a state pair *(q,q')* may require further merging of subsequent state pairs to obtain a deterministic solution; shared continuations of *q* and *q'* are folded up by such further merges. When this would end up in merging black and white states, the merging of *(q,q')* is discarded, and RPNI continues with the next candidate pair. (See [Dam05] for details.)

Fig.19 shows the system LTS computed by the synthesizer for our train example, with the following partition into equivalence classes:

$\pi$ = { {0,3,6,10,16}, {1,14,15}, {2,7,13}, {4}, {5}, {8}, {9}, {11,12} }

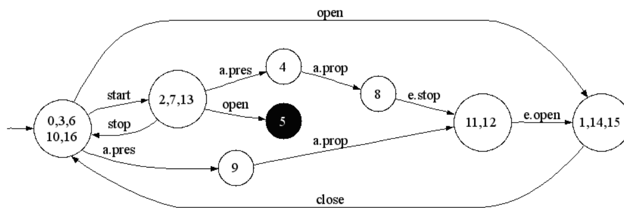



**Figure 19.** Synthesized system LTS for the train example

The equivalence relation used by this inductive algorithm shows the important role played by negative scenarios to avoid merging non-equivalent system states and derive correct generalizations. RPNI is guaranteed to find the correct system LTS when the input sample is rich enough [Onc92]; two distinct system states must be distinguished in the PTA by at least one continuation accepted from one and rejected from the other. When the input sample has no enough negative scenarios, RPNI tends to compute a poor generalization by merging non-equivalent system states.

To overcome this problem, the LTS synthesizer extends RPNI in two directions:

- Blue Fringe search: The search is made heuristic through an evaluation function that favors states sharing common continuations as first candidates for merging [Lan98].
- Interactive search: The synthesis process is made interactive through scenario questions asked by the synthesizer whenever a merged state gets new outgoing transitions [Dam05].

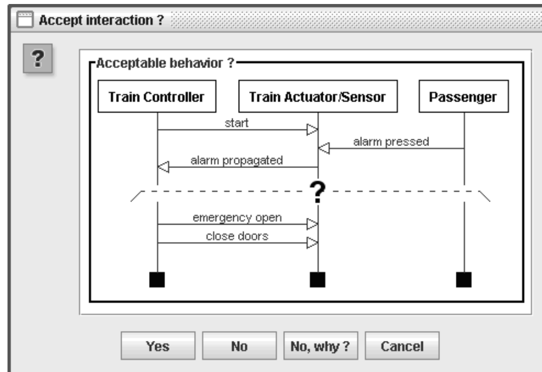



**Figure 20.** Scenario question generated during synthesis

To answer a scenario question, the user has just to accept or reject the new MSC scenario generated by the synthesizer. The answer results in confirming or discarding the current candidate state merge. Scenario questions provide a natural way of eliciting further positive and negative scenarios to enrich the scenario sample. Fig.20 shows a scenario question that can be rephrased as follows: "*if the train starts and a passenger presses the alarm button, may the controller then open the doors in emergency and close the doors afterwards?*". This scenario should be rejected as the train may not move with open doors.

There is a price to pay with this technique though. While interaction takes place in terms of simple end-user scenarios, and scenarios only, the number of scenario questions may sometimes become large for interaction-intensive applications with complex composite states - as experienced when applying the technique to non-trivial web applications.

This LTS synthesis technique was therefore recently extended to reduce the number of scenario questions significantly and produce a LTS model consistent with knowledge about the domain and about the goals of the target system [Dam06]. The general idea is to constrain the induction process in order to prune the inductive search space and, accordingly, the set of scenario questions. The constraints include:

a) state assertions generated along agent timelines;
b) LTS models of external components the system interacts with;
c) safety properties that capture system goals or domain properties.

Let us have a closer look at optimizations (a) and (c).

***Propagating fluents.*** Fluent definitions provide simple and natural domain descriptions to constrain induction. For example, the definition

fluent DoorsClosed = <{close doors}, {open doors, emergency open}> initially **true**

describes train door states as being either closed or open, and describes which event is responsible for which state change. To constrain the induction process, we compute the value of every fluent at each PTA state by symbolic execution. The PTA states are then decorated with the conjunction of such values. The pruning rule for constraining induction is to avoid *merging inconsistent states*, that is, states whose decoration has at least one fluent with different values. The specific equivalence relation here is thus the set of state pairs where both states have the same value for every fluent. The decoration of the merged state is simply inherited from the states being merged.

To compute PTA node decorations by symbolic execution, we use a simplified version of an algorithm described in [Dam05] to propagate fluent definitions forwards along paths of the PTA tree.

Fig.21 shows the result of propagating the values of fluent DoorsClosed, according to its above definition, along the PTA shown in Fig.18.

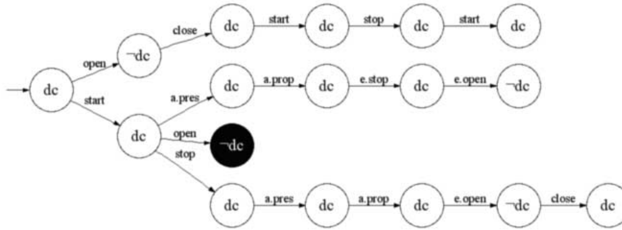



**Figure 21.** Propagating fluent values along a PTA (*dc* is a shorthand for *DoorsClosed*)

***Injecting goals and domain properties in the synthesis process.*** For goals or domain properties that can be formalized as safety properties, we may generate a property *tester* [Gia03], that is, a LTS extended with an error state such that every path leading to the error state violates the property. Consider, for example, the goal:

DoorsClosedWhileMoving = $\square$(Moving $\rightarrow$ DoorsClosed)

Fig.22 shows the tester LTS for this property (the error state is the black one). Any event sequence leading to the error state from the initial state corresponds to an undesired system behavior. In particular, the event sequence `<start, open>` corresponds to the initial negative scenario in Fig.17. As seen in Fig.22, the tester provides many more negative scenarios. Property testers can in fact provide potentially infinite classes of negative scenarios.

To constrain the induction process further, the PTA and the tester are traversed jointly in order to decorate each PTA state with the corresponding tester state. Fig. 23 shows the PTA decorated using the tester in Fig.22. The pruning rule for constraining the induction process is now to avoid merging states decorated with distinct states of the property tester. Two states will be considered for merging if they have the same property tester state.

This pruning technique has the additional benefit of ensuring that the synthesized system LTS satisfies the considered goal or domain property. A tester for a safety prop-

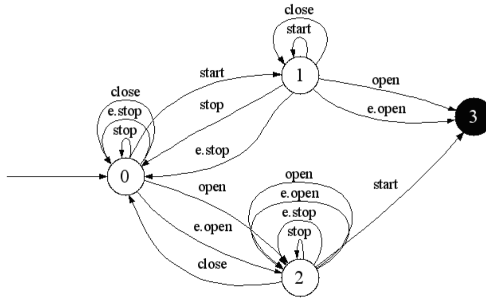



**Figure 22.** Tester LTS for the goal *DoorsClosedWhileMoving*

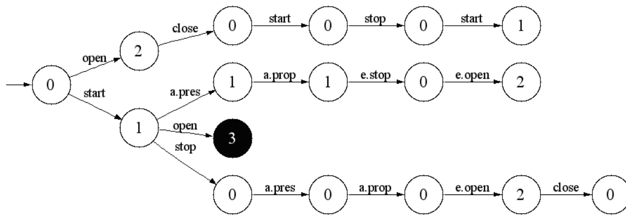



**Figure 23.** PTA decorated using the tester LTS from Fig. 22

erty is a canonical automaton, that is, minimal and deterministic [Gia03]. A bijection thus exists between states and continuations [Hop79]. In other words, two states are distinct if and only if there is at least one continuation to distinguish them. In the particular case of the tester LTS, two states are distinct if and only if they do not have the same set of continuations leading to the error state.

The question remains as to where these goals and domain properties are coming from. There are complementary answers to this:

- we can pick them up in the goal model, when they are available;
- we can get them systematically by asking the end-user the reason why a scenario is rejected as counterexample;
- we can infer some of them automatically by inductive inference from scenarios ([Lam98b], [Dam06]). In this case, the inferred property has to be validated by the user. If it turns to be inadequate, the user is asked to provide a counterexample scenario which will enrich the scenario collection.

## 11. Conclusion

It is important to verify that software applications implement their specifications correctly. However, do these specifications meet the software requirements (including non-functional ones)? Do these requirements meet the system's goals, and under realistic assumptions? Are these goals, requirements, and assumptions complete, adequate, and

consistent? These are critical, though still largely unexplored questions with many challenging issues for formal methods.

Rich models are essential to support the requirements engineering (RE) process. Such models must address multiple perspectives such as intentional, structural, responsibility, operational, and behavioral perspectives. They must cover the entire system, comprising both the software and its environment - made of humans, devices, other software, mother Nature, attackers, attackees, etc. They should also cover the current system-as-is, the system-to-be, and future evolutions. In our framework, such coverage is achieved through alternative subtrees in the goal AND/OR graph. Rich RE models should make alternative options explicit - such as alternative goal refinements, alternative agent assignments, alternative conflict resolutions, or alternative countermeasures to threats. They should support a seamless transition from high-level concerns to operational requirements.

Building such models is hard and critical. We should therefore be guided by methods that are systematic, incremental, supporting the analysis of partial models, and flexible to accommodate both top-down and bottom-up elaborations.

Goal-based reasoning is pivotal for model building and requirements elaboration, exploration and evaluation of alternatives, conflict management, anticipation of incidental or malicious behaviors, and optimization of behavior model synthesis.

Goal completeness is a key issue. It can be achieved through multiple means such as refinement checking to find out missing subgoals, obstacle and threat analysis to find countermeasure goals, or requirements animation [Tra04].

Declarative specifications play an important role in the RE process - in particular, for communicating with stakeholders and decision makers, for early reasoning about models, and for optimizing model synthesis.

In order to engineer highly reliable and secure systems, it is essential to start thinking methodically about these aspects as early as possible, that is, at requirements engineering time. We must be pessimistic from the beginning about the software and about its environment and anticipate all kinds of hazards, threats, and conflicts.

By discussing a variety of early analysis of RE models we hope we have been convincing on the benefits of a "multi-button" framework where semi-formal techniques are used for modeling, navigation, and traceability whereas formal techniques are used, when and where needed, for precise, incremental reasoning on mission-critical model portions. As suggested in this overview paper, goal-oriented models offer lots of opportunities for formal methods.

***Acknowledgement.*** Many of the ideas presented in this paper were developed over the years jointly with Robert Darimont, Emmanuel Letier, Christophe Damas, Anne Dardenne, Renaud De Landtsheer, David Janssens, Bernard Lambeau, Philippe Massonet, Christophe Ponsard, André Rifaut, Hung Tran Van, and Steve Fickas and his group at the University of Oregon. Warmest thanks to them all!

## References

[Bel76] T.E. Bell and T.A. Thayer, "Software Requirements: Are They Really a Problem?", Proc. ICSE-2: 2nd Intrnational Conference on Software Enginering, San Francisco, 1976, 61-68.

[Boe81] B.W. Boehm, Software Engineering Economics. Prentice-Hall, 1981.

[Bor93] A. Borgida, J. Mylopoulos and R. Reiter, "And Nothing Else Changes: The Frame Problem in Procedure Specifications", Proc. ICSE'93 - 15th International Conference on Software Engineering, Baltimore, May 1993

[Bro87] F.P. Brooks "No Silver Bullet: Essence and Accidents of Software Engineering". IEEE Computer, Vol. 20 No. 4, April 1987, pp. 10-19.

[Chu00] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, Non-functional requirements in software engineering. Kluwer Academic, Boston, 2000.

[Cla00] E.M. Clarke, S. Jha, and W. Marrero, "Verifying Security Protocols with Brutus", *ACM Trans. Software Engineering and Methodology* Vol. 9 No. 4, October 2000, 443-487.

[Dam05] C. Damas, B. Lambeau, P. Dupont and A. van Lamsweerde, "Generating Annotated Behavior Models from End-User Scenarios", *IEEE Transactions on Software Engineering*, Special Issue on Interaction and State-based Modeling, Vol. 31, No. 12, December 2005, 1056-1073.

[Dam06] C. Damas, B. Lambeau, and A. van Lamsweerde, "Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis", *14th ACM International Symp. on the Foundations of Software Engineering*, Portland (OR), Nov. 2006.

[Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", Science of Computer Programming, Vol. 20, 1993, 3-50.

[Dar96] R. Darimont and A. van Lamsweerde, Formal Refinement Patterns for Goal-Driven Requirements Elaboration. Proceedings FSE-4 - Fourth ACM Conference on the Foundations of Software Engineering, San Francisco, October 1996, 179-190.

[Del03] R. De Landtsheer, E. Letier and A. van Lamsweerde, "Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models", Requirements Engineering Journal Vol.9 No. 2, 104-120.

[Del05] R. De Landtsheer and A. van Lamsweerde, "Reasoning About Confidentiality at Requirements Engineering Time", *Proc. ESEC/FSE'05*, Lisbon, Portugal, Sept. 2005.

[Dij76] E.W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[Dos00] A. dos Santos, G. Vigna, and R. Kemmerer, "Security Testing of the Online Banking Service of a Large International Bank", *Proc. 1st Workshop on Security and Privacy in E-Commerce*, Nov. 2000.

[Dwy99] M.B. Dwyer, G.S. Avrunin and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification", Proc. ICSE-99: 21th Intl. Conference on Software Enginering, Los Angeles, 411-420.

[ESI96] European Software Institute, "European User Survey Analysis", Report USV_EUR 2.1, ESPITI Project, January 1996.

[Fag95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[Fea87] M. Feather, "Language Support for the Specification and Development of Composite Systems", ACM Trans. on Programming Languages and Systems 9(2), Apr. 87, 198-234.

[Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Fin94] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multiperspective Specifications", *IEEE Trans. on Software Engineering* Vol. 20 No. 8, 1994, 569-578.

[Gia03] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems", Proc. ESEC/FSE 2003, 10th European Software Engineering Conference, Helsinki, 2003.

[Ham01] J. Hammond, R. Rawlings, A. Hall, "Will it Work?", Proc. RE'01 - 5th Intl. IEEE Symp. on Requirements Engineering, Toronto, IEEE, 2001, 102-109.

[Hop79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Jac95] D. Jackson, "Structuring Z specifications with views", ACM Transactions on Software Engineering and Methodology, Vol. 4 No. 4, October 1995, 365-389.

[Jan06] D. Janssens and A. van Lamsweerde, Synthesizing Threat Models for Security Requirements Engineering, Département d'Ingénierie Informatique, Université Catholique de Louvain, August 2006.

[Jar98] M.Jarke and R. Kurki-Suonio (eds.), Special Issue on Scenario Management, IEEE Trans. on Sofware. Engineering, December 1998.

[Kem94] R. Kemmerer, C. Meadows, and J. Millen, "Three systems for cryptographic protocol analysis", *Journal of Cryptology* 7(2), 1994, 79-130.

[Kem03] R. Kemmerer, "Cybersecurity", Proc.ICSE'03 - *25th Intl. Conf. on Softw. engineering*, Portland, 2003, 705 - 715.

[Kru98] I. Kruger, R. Grosu, P. Scholz and M. Broy, From MSCs to Statecharts, *Proc. IFIP WG10.3/WG10.5 Intl. Workshop on Distributed and Parallel Embedded Systems* (SchloSS Eringerfeld, Germany), F. J. Rammig (ed.), Kluwer, 1998, 61-71.

[Lad95] P. Ladkin, in The Risks Digest, P. Neumann (ed.), ACM Software Engineering Notes 15, 1995.

[Lam98a] A. van Lamsweerde, R. Darimont and E. Letier, Managing Conflicts in Goal-Driven Rquirements Engineering, *IEEE Transactions on Software Engineering*, Special Issue on Managing Inconsistency in Software Development, Vol. 24 No. 11, November 1998, pp. 908 - 926.

[Lam98b] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1089-1114.

[Lam00a] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective". Keynote Paper, *Proceedings ICSE'2000 - International Conference on Software Engineering*, Limerick. IEEE Computer Society Press, June 2000, pp.5-19.

[Lam00b] A. van Lamsweerde and E. Letier, Handling Obstacles in Goal-Oriented Requirements Engineering, *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, Vol. 26, No. 10, October 2000.

[Lam04a] A. van Lamsweerde, "Elaborating Security Requirements by Construction of Intentional Anti-Models", *Proceedings of ICSE'04 - 26th International Conference on Software Engineering*, Edinburgh, May. 2004, ACM-IEEE , 148-157.

[Lam04b] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice", Invited Keynote Paper, *Proc. RE'04, 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004, 4-8.

[Lam07] A. van Lamsweerde, Requirements Engineering - *From System Goals to UML Models to Software Specifications*. Wiley, 2007.

[Lan98] K.J. Lang, B.A. Pearlmutter, and R.A. Price, "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm", In *Grammatical Inference*, Lecture Notes in Artificial Intelligence Nr. 1433, Springer-Verlag, 1998, 1-12.

[Let02a] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proceedings ICSE'2002 - 24th International Conference on Software Engineering*, Orlando, May 2002, 83-93.

[Let02b] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", *Proc. FSE'10: 10th ACM Symp. Foundations of Software Engineering*, Charleston, Nov. 2002.

[Let04] E. Letier and A. van Lamsweerde, "Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering", *Proc. FSE'04, 12th ACM International Symp. on the Foundations of Software Engineering*, Newport Beach (CA), Nov. 2004, 53-62.

[Let05] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and Control in Scenario-Based Requirements Analysis", *Proc. ICSE 2005 - 27th Intl. Conf. Software Engineering*, St. Louis, May 2005.

[Lev95] N. Leveson, *Safeware - System Safety and Computers*. Addison-Wesley, 1995.

[Low96] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *TACAS'96: Tools and Algorithms for Construction and Analysis of Systems*, 1996.

[Lut93] R.R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems", Proceedings RE'93 - First International Symposium on Requirements Engineering, San Diego, IEEE, 1993, 126-133.

[Mag00] J. Magee, N. Pryce, D. Giannakopoulou and J. Kramer, "Graphical Animation of Behavior Models", *Proc. ICSE'2000: 22nd Intl. Conf. on Software Engineering*, Limerick, May 2000, 499-508.

[Mag06] J. Magee and J Kramer, Concurrency - State Models & Java Programs. Second edition, Wiley, 2006.

[Mak01] E. Mäkinen and T. Systä, "MAS - An Interactive Synthesizer to Support Behavioral Modelling in UML", *Proc. ICSE'01 - Intl. Conf. Soft. Engineering*, Toronto, Canada, May 2001.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[Man96] Z. Manna and the STeP Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems", Proc. CAV'96 - 8th Intl. Conf. on Computer-Aided Verification, LNCS 1102, Springer-

Verlag, July 1996, 415-418.

[Myl92] Mylopoulos, J., Chung, L., Nixon, B., "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", IEEE Trans. on Sofware. Engineering, Vol. 18 No. 6, June 1992, pp. 483-497.

[Obj04] The Objectiver Toolset. http://www.objectiver.com.

[Onc92] J. Oncina and P. García, "Inferring Regular Languages in Polynomial Update Time", *In* N. Perez de la Blanca et al (Ed.), *Pattern Recognition and Image Analysis*, Vol. 1 Series in Machine Perception & Artificial Intelligence, World Scientific, 1992, 49-61.

[Par95] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", Science of Computer Programming, Vol. 25, 1995, 41-61.

[Pon04] Ch. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, H. Tran Van, "Early Verification and Validation of Mission-Critical Systems", Proc. FMICS'04, 9th International Workshop on Formal Methods for Industrial Critical Systems, Linz (Austria) Sept. 2004.

[Rob03] W.N. Robinson, S. Pawlowski and V. Volkov, "Requirements Interaction Management", *ACM Computing Surveys* Vol. 35 No. 2, June 2003, 132-190.

[Sta95] The Standish Group, "Software Chaos", http:// www.standishgroup.com/chaos.html.

[Tra04] H. Tran Van, A. van Lamsweerde, P. Massonet, Ch. Ponsard, "Goal-Oriented Requirements Animation",*Proc. RE'04, 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004, 218-228.

[Uch03] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioral Models from Scenarios", *IEEE Trans. Softw. Engineering*, 29(2), 2003, 99-115.

[Wal77] R. Waldinger, "Achieving Several Goals Simultaneously", in *Machine Intelligence*, Vol. 8, E. Elcock and D. Michie (Eds.), Ellis Horwood, 1977.

[Whi00] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios", *Proc. ICSE'2000: 22nd Intl. Conference on Software Engineering*, Limerick, 2000, 314-323.

[Yue87] K. Yue, "What Does It Mean to Say that a Specification is Complete?", Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design, Monterey, 1987.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Pervasive Verification of Distributed Real-Time Systems

Steffen KNAPP [1] and Wolfgang PAUL
*Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany*
e-mail: {sknapp,wjp}@wjpserver.cs.uni-sb.de

**Abstract.** In these lecture notes we outline for the first time in a single place a correctness proof for a distributed real-time system from the gate level to the computational model of a CASE tool.

**Keywords.** Model Stack, Pervasive Verification, Distributed System, Real-Time, Automotive

## 1. Introduction

The mission of the German Verisoft project [Verb] is (i) to develop tools and methods permitting the pervasive formal verification of entire computer systems including hardware, system software, communication systems and applications (ii) to demonstrate these methods and tools with examples of industrial complexity.

In the automotive subproject the following distributed real-time system is considered. The hardware consists of ECU's connected by a FlexRay-like bus [Fle]. The ECU's comprise a VAMP processor [BJK+03,DHP05] and a FlexRay-like interface. System software is a C0 compiler [LPP05] and an OSEKtime-like [OSE01b] operating system OLOS [Kna05] realized as a dialect of the generic operating system kernel CVM [GHLP05]. Applications are compiled C0 programs communicating via an FTCom-like [OSE01a] data structure. They are generated by a variant of the AutoFocus CASE tool; the computational model underlying this tool is a variant of communication automata. A pervasive correctness proof for this system was presented in the lectures of the second author at the summer school on 'Software System Reliability and Security' 2006 in Marktoberdorf. This survey paper contains the lecture notes.

In Section 2 we outline the specification of a DLX instruction set [HP96,MP00] including the handling of interrupts.

Using the VAMP processor [BJK+03] as an example we explain in Section 3 how to verify the hardware design of complex processors with internal and external interrupts. The resulting correctness proofs are based on the scheduling functions introduced in [SH98,MP00].

[1] Work partially funded by the International Max Planck Research School for Computer Science (IMPRS) and the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Section 4 deals with a generic device theory. We show how to specify devices and how to integrate these specifications into the instruction set architecture of a processor.

In Section 5 we extend the VAMP processor design with memory management units (MMUs). This gives hardware support for multi processing operating system kernels and for virtual machine simulation [DHP05][2].

In Section 6 we survey a formal correctness proof for a compiler from the C0 programming language [LPP05,Pet06,Lei06] to the DLX instruction set. In a nutshell C0 is PASCAL [HW73] with C syntax.

In Section 7 we extend the C0 language. We permit portions of inline assembler code and call the resulting language $C0_A$. Using the allocation function of the compiler from Section 6 we can define the semantics of $C0_A$ programs in a natural way.

In Section 8 we describe the semantics of the generic operating system kernel CVM [GHLP05], which stands for communicating virtual machines. The programmer sees a so called *abstract* kernel and a set of user processes. The user processes are virtual DLX machines. The abstract kernel is a C0 program that makes use of certain so called CVM primitives. These primitives allow the transport of data between kernel and user processes. The semantics of the primitives can be specified in the parallel user model.

The implementation of a CVM kernel requires linking some extra code to the abstract kernel as described in Section 9. This results in a so called *concrete* kernel. The concrete kernel necessarily contains inline assembler code, because machine registers and user processes are simply not visible in C0 variables alone. The correctness proof hinges on the virtual machine simulation from Section 5, the compiler correctness proof form Section 6 and on the inline assembler semantics from Section 7.

Next we would like to instantiate the abstract CVM kernel with an OSEKtime-like operating system kernel called OLOS [Kna05]. User processes running under OLOS will be C0 programs. These programs communicate via FTCom-like message buffers with processes running on the same *and* on remote processors. While the machinery available at the end of Section 9 permits effortlessly to define the application programmers model, for a pervasive correctness proof of the entire distributed system we lack an important ingredient: A correctness proof for a FlexRay-like communication system between processors.

Since the ECUs are running with local oscillators of almost but not exactly equal clock frequency, we cannot guarantee that set up and hold times of registers are respected when data is being transmitted between ECUs. In such situations serial interfaces are used. In Section 11 we review a correctness proof for a serial interface from [BBG$^+$05].

In Section 12 we construct I/O devices called *f-interfaces*, consisting among other things of message buffers, serial interfaces, and local timers. An ECU consists of a processor together with such an interface. In time triggered protocols like FlexRay, ECUs communicate in fixed time slots; in the simplest case via a single bus. In each time slot one ECU is allowed to broadcast its message buffers and the other ECU's must remain quiet. This only works, if local timers on the ECUs are kept roughly synchronized. The implementation and correctness proof of a non fault tolerant clock synchronization algorithm –built on top of the serial interfaces of Section 11– is therefore part of Section 12. Extension of this section to the fault tolerant case is future work and has two parts: (i) clock synchronization in the fault tolerant case; this is an extremely well studied

[2]In real-time systems the virtual machine simulation is done in a restricted way such that no page faults occur.

problem [Sch87,Rus94] (ii) a startup algorithm for the fault tolerant case. In view of results reported in [SK06], this might require some modifications in the start-up algorithm from the FlexRay standard.

In Section 13 we use techniques from [HIP05] to integrate the f-interfaces into the ISA (instruction set architecture) of the processor. Due to the (external) timer interrupts we run into a problem which is both surprising and not so easy to overcome: Timer interrupts occur in fixed time intervals. It is trivial to determine on the hardware level in which cycle such an interrupt occurs. We have to define on the ISA level the corresponding instruction that gets interrupted. This can inherently not be done on the ISA level alone: The execution time of an instruction depends on cache hits and cache misses, but the memory hierarchy is invisible on the ISA level. On the pure ISA level we end up with a nondeterministic model of computation.

We formalize the nondeterminism by oracle inputs that indicate for each instruction if it is interrupted by a timer interrupt or not. The oracle inputs are determined as a byproduct of the processor correctness proof. This is intuitively plausible: If one is allowed to look inside the hardware at the register transfer language (RTL) level, then the occurrence of timer interrupts becomes deterministic. Technically we achieve this with the help of the scheduling functions introduced in Section 3.

In Section 14 we show how to combine classical program correctness proofs (on the ISA level), worst case execution time (WCET) analysis on the RTL level and hardware correctness proofs into pervasive correctness proofs for real-time system from the gate level to the ISA level. The results of Sections 11 to 14 are from [KP06].

In Section 15 we define the distributed OLOS model (D-OLOS) from [Kna05]: The realtime operating system OLOS is running on every ECU of the distributed system. User processes are compiled C0 programs. Using operating system calls they can communicate by accessing an FTCom-like data structure on their local ECU.

A pervasive correctness proof for the implementation of D-OLOS outlined in Section 16 is based on the correctness of the CVM implementation from Section 8, the compiler correctness from Section 6 and the results from Section 14.

In Section 17 we introduce the automaton-theoretic computational model of a CASE tool called AutoFocus task model (AFTM).

Based on results from [BBG$^+$06] we show in Section 18 how to simulate this model by D-OLOS.

## 2. Specifying an Instruction Set Architecture (ISA)

For bit strings $a = a[n-1:0] \in \{0,1\}^n$ we denote the natural number with binary representation $a$ by:

$$\langle a \rangle = \sum_{i=0}^{n} a_i \cdot 2^i$$

For numbers $x \in \{0, \ldots, 2^n - 1\}$ the binary representation of $x$ of length $n$ is the bit string $bin_n(x) \in \{0,1\}^n$ satisfying:

$$\langle bin_n(x) \rangle = x$$

The $n$ bit binary addition function $+_n : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ is defineded by:

$$a +_n b = bin_n(\langle a \rangle + \langle b \rangle \bmod 2^n)$$

For bits $x$ and natural numbers $n$ we define $x^n$ as the string obtained by concatenating $x$ exactly $n$ times with itself:

$$x^n = x \circ \ldots \circ x$$

### *2.1. Configurations and Auxiliary Concepts*

In this section we outline how to formally specify the DLX instruction set architecture (ISA). Processor configurations $d$ have the following components:

1. The $d.R$ component stores the current value of register $R \in \{0,1\}^{32}$. For this paper, the most relevant registers are: The program counter $pc$, the delayed PC[3] $dpc$, the general purpose registers $gpr[x]$ with $x \in \{0,1\}^5$ and the status register $sr$ containing the mask bits for the interrupts.
2. The byte addressable memory $d.m : A \rightarrow \{0,1\}^8$ where the set of addresses $A \subset \{0,1\}^{32}$ usually has the form $A = \{a \mid \langle a \rangle \leq d.b\}$ for some maximal available memory byte address $d.b$. The content of the memory at byte address $a$ is given by $d.m(a)$.

The maximal available address $d.b$ does not change during an ISA computation. Therefore it is rather treated as a parameter of the model than as a component of a configuration. We will later on partition memory into pages of $4K$ bytes. We assume that $d.b$ is a multiple of some page size

$$d.b = d.ptl \cdot 4K$$

where $d.ptl$ is a mnemonic for the last index of page tables (detailed in Section 5). For addresses $a$, memories $m$ and natural numbers $x$ we denote by $m_x(a)$ the concatenation of the memory elements from address $a$ to address $a + x - 1$ in little endian order:

$$m_x(a) = m(a + x - 1) \circ \ldots \circ m(a)$$

The instruction executed in configuration $d$, denoted by $I(d)$, is the memory word addressed by the delayed PC:

$$I(d) = d.m_4(d.dpc)$$

The six high-order bits of the instruction word constitute the opcode $opc$:

$$opc(d) = I(d)[31:26]$$

[3]The delayed PC is used to specify the delayed branch mechanism detailed in [MP00].

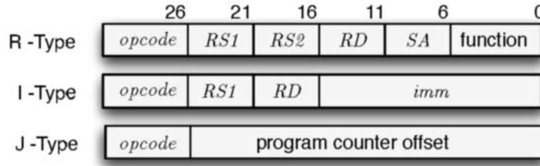



**Figure 1.** Instruction Types

Instruction decoding can easily be formalized by predicates on $I(d)$. In some cases it suffices to inspect the opcode only. The current instruction is for instance a 'load word' $lw$ instruction if the opcode equals 100011:

$$lw(d) \Leftrightarrow opc(d) = 100011$$

DLX instructions come in three instruction types as shown in Figure 1. The type of an instruction defines how the bits of the instruction outside the opcode are interpreted. The occurrence of an R-type instruction, e.g. an add or a subtract instruction, is for instance specified by:

$$rtype(d) \Leftrightarrow opc(d) = 000000$$

Definitions of I-type and J-type instructions are slightly more complex. Depending on the instruction type, certain fields have different positions within the instruction. For the register 'destination' operand $RD$ we have for instance

$$RD(d) = \begin{cases} I(d)[20:16] & itype(d) \\ I(d)[15:11] & \text{otherwise} \end{cases}$$

The effective address $ea$ of load / store operations is computed as the sum of (i) the content of the register addressed by the $RS1$ field $d.gpr(RS1(d))$ and (ii) the immediate field $imm(d) = I(d)[15:0]$. The addition is performed modulo $2^{32}$ with two's complement arithmetic. Formally, we define the sign extension of the immediate constant by:

$$sxt(imm(d)) = imm(d)[15]^{16} \circ imm(d)$$

This turns the immediate constant into a 32-bit constant while preserving the value as a two's complement number. It is like adding leading zeros to a natural number. The effective address is defined as:

$$ea(d) = d.gpr(RS1(d)) +_{32} sxt(imm(d))$$

This definition is possible since $n$ bit two's complement numbers and $n$ bit binary numbers have the same value modulo $2^n$. For details see e.g. Chapter 2 of [MP00].

### 2.2. Basic Instruction Set

With the above definitions in place we specify the next configuration $d'$, i.e. the configuration after execution of $I(d)$. This obviously formalizes the instruction set.

In the definition of $d'$ we split cases depending on the instruction to be executed. As an example we specify the next configuration for a load word and a store word instruction.

The main effect of a load word instruction is that the general purpose register addressed by the $RD$ field is updated with the memory word addressed by the effective address $ea$:

$$d'.gpr(RD(d)) = d.m_4(ea(d))$$

The PC is incremented by four in 32-bit binary arithmetic and the old PC is copied into the delayed PC:

$$\begin{aligned} d'.pc &= d.pc +_{32} bin_{32}(4) \\ d'.dpc &= d.pc \end{aligned}$$

This part of the definition is identical for all instructions except control instructions. Components that are not changed have to be specified, too:

$$\begin{aligned} d'.m &= d.m \\ d'.gpr(x) &= d.gpr(x) \qquad \text{for } x \neq RD(d) \\ d'.sr &= d.sr \end{aligned}$$

The main effect of store word instructions is that the general purpose register content addressed by $RD$ is copied into the memory word addressed by $ea$:

$$d'.m_4(ea(d)) = d.gpr(RD(d))$$

Completing this definition for all instructions results in the the definition of a DLX next state function:

$$d' = \delta_D(d)$$

### 2.3. Dealing with Interrupts

Interrupts are triggered by interrupt event signals that might be internally generated (like illegal instruction, misalignment, and overflow) or externally generated (like reset and timer interrupt). Interrupts are numbered using indices $j \in \{0, \ldots, 31\}$. We classify the set of these indices in two categories:

1. maskable / not maskable. The set of indices of maskable interrupts is given by $M$.
2. external / internal. The set of indices of external interrupts is given by $E$.

We denote external event signals by $eev[j]$ with $j \in E$ and we denote internal event signals by $iev[j]$ with $j \notin E$. We gather the external event signals into a vector $eev$ and the internal event signals into a vector $iev$.

Formally these signals must be treated in a very different way. Whether an internal event signal $iev[j]$ is activated in configuration $d$ is determined only by the configuration. For instance if we use $j = 1$ for the illegal instruction interrupt and $LI \subset \{0,1\}^{32}$ is the set of bit patterns for that $d'$ is defined if $I(d) \in LI$, then:

$$iev(d)[1] \Leftrightarrow I(d) \notin LI$$

Thus the vector of internal event signals is a function $iev(d)$ of the current processor configuration $d$. In contrast, external interrupts are external inputs to the next state function. Therefore we get a new next state function:

$$d' = \delta_D(d, eev)$$

The cause vector $ca$ of all event signals is a function of the processor configuration $d$ and the external input $eev$:

$$ca(d, eev)[j] = \begin{cases} eev[j] & j \in E \\ iev(d)[j] & \text{otherwise} \end{cases}$$

The masked cause vector $mca$ is computed from $ca$ with the help of the interrupt mask stored in the status register. If interrupt $j$ is maskable and $sr[j] = 0$, then $j$ is masked out:

$$mca(d, eev)[j] = \begin{cases} ca(d, eev)[j] \wedge d.sr[j] & j \in M \\ ca(d, eev)[j] & \text{otherwise} \end{cases}$$

If any one of the masked cause bits is on, the jump to interrupt service routine (JISR) bit is turned on:

$$JISR(d, eev) = \bigvee_j mca(d, eev[j])$$

If this occurs, many things happen, e.g. the PCs are forced to point to the start addresses of the interrupt service routine. We assume it starts at (binary) address 0:

$$d'.dpc = bin_{32}(0)$$
$$d'.pc = bin_{32}(4)$$

All maskable interrupts are masked and the masked cause register is saved into a new exception cause register:

$$d'.sr = 0^{32}$$
$$d'.eca = mca(d, eev)$$

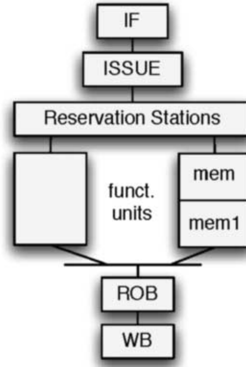



**Figure 2.** Processor Pipeline

Since interrupt lines might become active simultaneously it is important to know the smallest index of an active bit of $mca$. This index is called the *interrupt level* and specifies the interrupts of highest priority that will be serviced immediately:

$$il(d, eev) = \min\{j \mid mca(d, eev)[j] = 1\}$$

Auxiliary data for the intended interrupt handler is stored in an exception data register $edata$. We only specify the new content for the case of trap instructions. In the DLX instruction set the trap instruction has J-type format with opcode $111110$. We give the trap instruction interrupt event line 5:

$$iev(d)[5] \Leftrightarrow opc(d) = 111110$$

If this event line is active and no line with higher priority is active, then a trap interrupt occurs:

$$trap(d, eev) \Leftrightarrow il(d, eev) = 5$$

In case of a trap interrupt, the sign extended (26 bit) immediate constant is saved in the exception data register

$$trap(d, eev) \Rightarrow d'.edata = imm(d)[25]^6 \circ imm(d)$$

A complete definition of the interrupt mechanism is given in Chapter 5 of [MP00].

## 3. Processor Correctness

### 3.1. Processor Hardware Model

The processor hardware is specified in a hardware model. A hardware configurations $h$ consists of $n$ bit registers $h.R \in \{0,1\}^n$ and $(a \times d)$-RAMs $h.r : \{0,1\}^a \to \{0,1\}^d$.

Registers and RAMs are connected by Boolean circuits with the usual semantics from switching theory.

We denote the value of a signal $s$ in configuration $h$ by $s(h)$. The hardware transition function $\delta_H$ depends on external inputs $ein$. It maps a hardware configuration $h$ to the hardware configuration $h' = \delta_H(h, ein)$ after the next clock cycle. We define for a register $R$ with clock enable signal $Rce$ and input $Rin$:

$$h'.R = \begin{cases} Rin(h) & Rce(h) = 1 \\ h.R & \text{otherwise} \end{cases}$$

Given a RAM $r$ with address signal $addr$, data input $Din$ and a write signal $w$ we define:

$$h'.r(x) = \begin{cases} Din(h) & x = addr(h) \wedge w(h) \\ h.r(x) & \text{otherwise} \end{cases}$$

Hardware computations are defined in the usual way as sequences of configurations $h^0, h^1, \ldots$. A superscript $t$ in this model is always read as 'during cycle $t$'. Hardware computations must satisfy for all cycles $t$:

$$h^{t+1} = \delta_H(h^t, ein^t)$$

Processor correctness theorems state, that hardware defined in this model simulates in some sense an ISA next state function $\delta_D$ as defined in the previous sections.

### 3.2. Scheduling Functions

The processor correctness proofs considered here hinge on the concept of scheduling functions $s$. The hardware of pipelined processors consists of many stages $k$, e.g. fetch stage, issue stage, reservation stations, reorder buffer, write back stage, etc. (see Figure. 2). Stages can be full or empty due to pipeline bubbles. The hardware keeps track of this with the help of full bits $full_k$ for each stage as defined in [MP00]. Recall that $full_k(h^t)$ is the value of the full bit in cycle $t$. We use the shorthand $full_k^t$. Note that the fetch state is always full, i.e. $\forall t : full_0^t = 1$.

For hardware cycles $t$ and stages $k$ that are full during cycle $t$, i.e. such that $full_k^t$ holds, the value $s(k, t)$ of the scheduling function is the index $i$ of the instruction that is in stage $k$ during cycle $t$. If the stage is not full, it is the index of the instruction that was in stage $k$ in the last cycle before $t$ when the stage was full. Initially $s(0, 0) = 0$ holds.

In the formal definition of scheduling functions we use an extremely simple idea: Imagine that the hardware has registers that can hold integers of arbitrary size. Augment each stage with such a register and store in it the index of the instruction currently being executed in that stage. These indices are computed exactly as the tags in a Tomasulo scheduler. The only difference is that the indices have unbounded size because we want to count up to arbitrarily large indices. In real hardware this is not possible and not necessary. Nevertheless, in an abstract mathematical model there is no problem to do this.

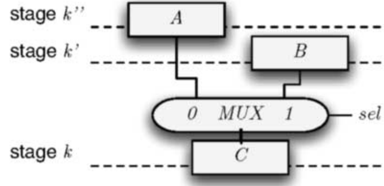



**Figure 3.** Scheduling Functions

Each stage $k$ of the processors under consideration has an update enable signal $ue_k$. Stage $k$ gets new data in cycle $t$ if the update enable signal $ue_k$ was on in cycle $t-1$. We fetch instructions in order and hence define for the instruction fetch stage $IF$:

$$s(IF,t) = \begin{cases} s(IF,t-1)+1 & ue_{IF}^{t-1} \\ s(IF,t-1) & \text{otherwise} \end{cases}$$

In general, a stage $k$ can get data belonging to a new instruction from one or more stages $k'$. Examples where more than one predecessor stage $k'$ exists for a stage $k$ are: (i) cycles in the data path of a floating point unit performing iterative division or (ii) the producer registers feeding on the common data bus of a Tomasulo scheduler. In this situation one must define for each stage $k$ a predicate $trans(k',k,t)$ indicating that in cycle $t$ data are transmitted from stage $k'$ to stage $k$. In the example of Figure 3 we use the select signal $sel$ of the multiplexer and define:

$$trans(k',k,t) = ue_k^t \wedge sel^t$$

If $trans(k',k,t-1)$ holds for some $k'$, then we set $s(k,t) = s(k',t-1)$ for that $k'$. Otherwise $s(k,t) = s(k,t-1)$.

### 3.3. Naive Simulation Relations

For ECUs we first consider a 'naive' simulation relation $sim(d,h)$ between ISA configurations $d$ and hardware configurations $h$. We require that user-visible processor registers $R$ have identical values:

$$h.R = d.R$$

For the addresses $a$ in the processor we would like to make a similar definition, but this does not work, because the user-visible processor memory is simulated in the hardware by a memory system consisting among others of an instruction cache $icache$, a data cache $dcache$ and a user main memory $mainm$. Thus there is a quite nontrivial function $m(h) : A \rightarrow \{0,1\}^8$ specifying the memory simulated by the memory system. One can define this functions in the following way: Imagine you apply in configuration $h$ at the memory interface (either at the $icache$ or at the $dcache$) address $a$. Considering a hit in the instruction cache, i.e. $ihit(h,a) = 1$, the $icache$ would return $icache(h,a)$.

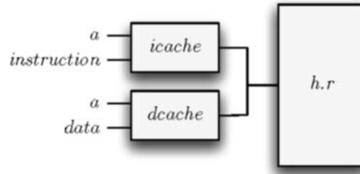



**Figure 4.** Memory System

Similarly, considering a hit in the data cache $dhit(h,a) = 1$ the $dcache$ would return $dcache(h,a)$. Then we define:[4]

$$m(h)(a) = \begin{cases} icache(h,a) & ihit(h,a) \\ dcache(h,a) & dhit(h,a) \\ h.mainm(a) & \text{otherwise} \end{cases}$$

Using this definition we additionally require in the simulation relation $sim(d,h)$ for all addresses $a \in A$:

$$m(h)(a) = d.m(a)$$

In a pipelined machine this simulation relation almost never holds, because in one cycle different hardware stages $k$ usually hold data from different ISA configurations; after all this is the very idea of pipelining. There is however an important exception: When the pipeline is drained, i.e. all hardware stages except the instruction fetch stage are empty:

$$drained(h^t) \Leftrightarrow \forall k \neq IF : \neg full_k^t$$

This happens to be the case after interrupts, in particular initially after reset.

### 3.4. Basic Processor Correctness Theorem

To begin with we ignore the external interrupts event signals (which brings us formally back to ISA computations defined by $d^{i+1} = \delta_D(d^i)$). Figure 2 shows in simplified form the stages of a processor with out of order processing and a Tomasulo scheduler.

Each user-visible register $d.R$ of the processor has a counter part $h.R$ belonging to some stage $k = stage(R)$ of the hardware. If the processor would have only registers $R$ and no memory, we could show by induction over $t$ that for all cycles $t$ and stages $k$: If $k = stage(R)$, then the value $h^t.R$ of the hardware register $R$ in cycle $t$ is the value $d^{s(k,t)}.R$ of the ISA register $R$ for the instruction scheduled in stage $k$ in cycle $t$:

$$h^t.R = d^{s(k,t)}.R$$

[4] In the processors under consideration the caches snoop on each other. Hence the data of address $a$ is only in at most one cache [Bey05,BJK+03]

For the memory one has to consider the memory unit of the processor consisting of two stages $mem$ and $mem1$. Stage $mem$ contains hardware for the computation of the effective address. The memory $m(h^t)$ that is simulated by the memory hierarchy of the hardware in cycle $t$, is identical with the ISA memory $d^{s(mem1,t)}.m$ for the instruction scheduled in stage $mem1$ in cycle $t$:

$$m(h^t) = d^{s(mem1,t)}.m$$

We summarize the above by stating a basic processor correctness theorem. It assumes that initially the pipe is drained and that the simulation relation between the first hardware configuration $h^0$ and the first ISA configuration $d^0$ holds.

**Theorem 1 (Processor Correctness)** *Assume that $drained(h^0)$ and $sim(d^0, h^0)$ holds. Then for all t, for all stages $k$ and for all registers $R$ with $stage(R) = k$:*

$$h^t.R = d^{s(k,t)}.R$$
$$m(h^t) = d^{s(mem1,t)}.m$$

Such theorems are proven by induction over $t$. For complex processors this requires hundreds of pages of paper and pencil proofs (see [MP00]). A formal correctness proof is described in [Bey05,BJK$^+$03].

### 3.5. Dealing with External Interrupts

External interrupts complicate things only slightly. The hardware now has external inputs $heev$ that we call the hardware interrupt event signals. Their value in hardware cycle $t$ is $heev^t$. We have to construct from them a sequence $eev^i$ of external ISA interrupt event signals such that the hardware simulates an ISA computation satisfying $d^{i+1} = \delta_D(d^i, eev^i)$.

In order to support precise interrupts the processor hardware usually samples interrupt event signals in the write back stage WB (see Chapter 5 of [MP00]). Since the write back stage is the last stage in the pipeline it cannot be stalled. Thus for every instruction $i$ there is exactly one cycle $t = WB(i)$ such that $s(WB, t) = i \wedge full^t_{WB}$. The external ISA event signal observed by instruction $i$ is therefore:

$$eev^i = heev^{WB(i)}$$

Note that a hardware event signal $heev^t$ is not visible to the ISA computation if the write back stage in cycle $t$ is empty. With this new definition of the ISA computation Theorem 1 still holds. More details regarding a formal processor correctness proof dealing with external interrupts are given in [Bey05,Dal06].

## 4. Device Theory

### 4.1. Device Configurations

This section basically covers the device independent part of [HIP05]. In memory mapped I/O processors communicate with devices by read and write accesses to certain word ad-

dresses $x$ called *I/O ports*. In our treatment these addresses will be above the addresses in the processor memory. For a hardware designer who integrates a device into a processor the device therefore should better look in many respects like an ordinary RAM. However, a device has in general more state than is visible in the I/O ports. Thus configurations of devices $f$ with $N$ I/O ports have the following components:

- A port RAM $f.m$. We assume that the RAM is byte addressable providing $P$ bytes, i.e. $f.m : \{0,1\}^p \rightarrow \{0,1\}^8$ with $p = \lceil \log P \rceil$.
- An 'internal' state $f.Z$.

Hardware devices take inputs from and produce output to the processor side and to the outside world respectively. Inputs from the processor side are like inputs for the RAM and consist of: Data input $din$, address $addr$, and write signal $fw$. Outputs to the processor side consists of data output $dout$ (like in a RAM) and an external hardware interrupt event signal $heev$. Inputs $fdin$ from and outputs $fdout$ to the outside world are device dependent: Network devices have inputs and outputs, monitors produce only outputs, keyboards take only inputs, disks neither produce outputs nor consume inputs.

I/O ports can be roughly divided in three categories: (i) control ports are only written from the processor side, (ii) status ports are only read by the processor side and (iii) data ports can be written or read both from the processor side and from the device side. Thus we have to deal with the classical synchronization issues of shared memory.

In order to be able to use existing hardware correctness proofs for processors alone, we split the hardware $h$ into a processor component $h.p$ and make the device configuration $f$ a component $h.f$ of the hardware configuration.

We postulate, that for each word address $addr$ of the device there is a device specific hardware predicate $hquiet(f, addr)$ acting like a semaphore. It indicates that the 4 ports belonging to that address are presently not being accessed from the device side and hence it is safe to access them from the processor side like ordinary RAM. We define for reading out the port RAM:

$$dout(h, addr) = h.f.m(addr)$$

At quiet word addresses $x$ the port RAM behaves like a RAM accessible only by the processor side:

$$\forall x : hquiet(h.f, x) \Rightarrow h'.f.m_4(x) = \begin{cases} din(h) & x = addr(h) \wedge fw(h) \\ h.f.m_4(x) & \text{otherwise} \end{cases}$$

When a data port is not quiet, it can be read or modified by the device side in a device specific way. The effect of writing a port that is not $quiet$ is left undefined. The processor side usually learns about changes in the quiet predicate either by an interrupt from the device or by polling a status register. We do not consider polling here.[5]

[5]A reader experienced in hardware design will observe that our devices are unusually fast: They update a port in a single cycle of the processor hardware. Devices are usually slower and thus require a busy signal indicating if a read or write access is in progress. Extending the above definitions in this way poses no big difficulties.

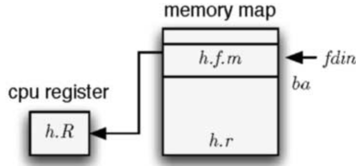



**Figure 5.** Memory Mapped IO

### *4.2. Integrating Devices*

Integration of a device into a memory system is an exercise in hardware design. The device is placed at some base address $ba$ into the processors (byte addressable) memory. An address decoder decides on read or write accesses whether the device is addressed by a load word or store word instruction ($ea \in \{ba, \ldots, ba + P - 1\}$). If the base address is a multiple of the page size $4K$ and if the device occupies exactly one page of memory, then the address decoder simply performs the test:

$$ba[31:12] =^? ea[31:12]$$

If device $f$ is accessed by a store word instruction, then write signal $fw$ is activated. If the device is accessed by a load word instruction, then the output enable signal of a driver between $dout$ and some bus in the processors memory system is enabled. The cache system must be designed in a way that it does not cache accesses to I/O ports.

### *4.3. ISA with Devices*

The assembler programmer sees a system as shown in Figure 5. It is a distributed system because the non quiet ports of the device can change in a device specific way while the processor is working. Configurations have the form $ecu = (ecu.d, ecu.f)$ where $ecu.d$ is an ISA processor configuration and $ecu.f$ is the device configuration. Component $ecu.f$ might have the same form as the device configuration from the hardware model or it might be more abstract. In the assembler model the programmer should have some means to keep track of the quiet status of ports. Thus, the hardware predicate $hquiet(h.f, addr)$ needs a device specific assembler level counter part $quiet(ecu.f, addr)$.

As we have argued in the introduction, for a processor with a device the occurrence of external interrupts is inherently nondeterministic in an assembler level model. We model this nondeterminism by an oracle input $eev$ that is used in the next state computation of the processor component $ecu'.d$ as explained in Section 3.5. In the case of accesses to the I/O ports the next processor state will depend on the device state, too. Thus we will define an extension of the old next state function $\delta_D$:

$$ecu'.d = \delta_D(ecu.d, ecu.f, eev)$$

The extension concerns load word instructions whose effective address is an I/O port and of course the occurrence of interrupts:

$$\neg JISR(ecu.d, eev) \wedge ea(ecu.d) = ba + 4 \cdot addr \wedge quiet(ecu.f, addr)$$
$$\wedge\; lw(ecu.d) \Rightarrow ecu'.d.gpr(RD(ecu.d)) = ecu.f.m(addr)$$

Moreover we can specify in a device independent way, that quiet word addresses $x$ of the port RAM behave like processor memory:

$$\forall x : quiet(ecu.f, x) \wedge \neg JISR(ecu.d, eev) \Rightarrow$$
$$ecu'.f.m_4(x) = \begin{cases} ecu.d.gpr(RD(ecu.d)) & sw(ecu.d) \wedge ea(ecu.d) = ba + 4 \cdot x \\ ecu.f.m_4(x) & \text{otherwise} \end{cases}$$

The remaining portions of the definition of $ecu'.f$ are device specific. We will come back to this point in Section 13.

### 4.4. Processor Correctness Theorem with Devices

Using the machinery already in place the extensions to the hardware correctness proof are remarkably easy as long as computations only access quiet I/O ports and the quiet predicate is stable for all ports. If we place in the hardware the devices port RAM parallel to the normal memory system, then we can use the same scheduling functions as for the memory:

**Theorem 2 (Processor Correctness with Devices)**

$$h^t.p.R = ecu^{s(k,t)}.d.R$$
$$m(h^t.p) = ecu^{s(mem1,t)}.d.m$$
$$h^t.f.m = ecu^{s(mem1,t)}.f.m$$

The external interrupt from the device will need device specific arguments. The hardware correctness proof works with the oracle inputs $eev^i$ obtained from the hardware event signal $heev^t$ by the translation from Section 3.5:

$$eev^i = heev^{WB(i)}$$

## 5. Memory Management

### 5.1. Address Translation, Physical Machines and Virtual Machines

Physical machines consist of a processor operating on physical memory and on swap memory. Configurations $d$ of physical machines have components $d.R$ for processor registers $R$, $d.m$ for the physical memory, and $d.sm$ for the swap memory. The physical ma-

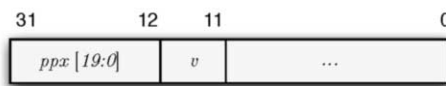



**Figure 6.** Page Table Entry

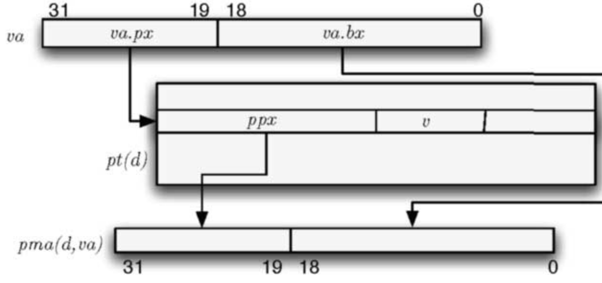



**Figure 7.** Address Translation

chine has several special purpose registers, e.g. the mode register $mode$, the page table origin $pto$, and the page table length $ptl$.

In system mode, i.e. if $d.mode = 0$, the physical machine operates like the basic processor model from Section 2 with extra registers.

In user mode, i.e. if $d.mode = 1$, the physical machine *emulates* the basic processor model from Section 2 using the page table for address translation. The simulated machine is called a *virtual machine*. The addresses used in by the virtual machines are called virtual addresses. We keep the notation $d$ for configurations of the physical machine and we denote configurations of the virtual machine by $vm$. Virtual addresses $va$ are split into a page index $va.px = va[31 : 12]$ and a byte index $va.bx = va[11 : 0]$. Thus the pages size is $2^{12} = 4K$ bytes.

In user mode an access to memory address $va$ is subject to address translation. It either cause a page fault or it is redirected to the translated physical memory address $pma(d, va)$. The result of address translation depends on the content of the *page table*, a region of the physical memory starting at address $d.pto \cdot 4K$ with $d.ptl + 1$ entries of four bytes width.[6]

Page table entries have a length of four bytes. The page table entry address for virtual address $va$ is defined as $ptea(d, va) = d.pto \cdot 4K + 4 \cdot va.px$ and the page table entry of $va$ is defined as $pte(d, va) = d.m_4(ptea(d, va))$. For our purposes a page table entry consists of two components as shown in Figure 6: The physical page index $ppx(d, va) = pte(d, va)[31 : 12]$ and the valid bit $v(d, va) = pte(d, va)[11]$.

Being in user mode and accessing memory address $va$, a page fault signals if the page index exceeds the page table length, $va.px > d.ptl$ or if the page table entry is not valid, $v(d, va) = 0$. On page fault the page fault handler, an interrupt service routine, is invoked.

If no page fault is generated, the access is performed on the (translated) physical memory address $pma(d, va)$ defined as the concatenation of the physical page index and the byte index (see Figure 7):

$$pma(d, va) = ppx(d, va) \circ va.bx$$

Notice that the complete definition of a physical machine model involves the specification of the effect of a page fault handler. In pervasive system verification there exists a

[6] The '+1' in this definition is awkward. It dates back to very old architectures. The page table length is usually a power of two, hence a bit in the page table length register is saved.

model between the physical machine and the hardware: A processor with a disk as an I/O device. In this model we can show that swap memory is a proper abstracting by proving the correctness of the page fault handler. For details see [HIP05]. Real-time systems, as being considered here, have no disks and are programmed such that page faults do not occur; thus the omission of these details will not hurt us later.

### 5.2. Virtual Memory Simulation

Physical machines with appropriate page fault handlers can simulate virtual machines. For a simple page fault handler, virtual memory is stored on the swap memory of the physical machine and the physical memory acts as a write back cache. In addition to the architecturally defined physical memory address $pma(d, va)$, the page fault handler maintains a swap memory address function $sma(d, va)$. On page faults that do not violate the page table length check, the handler selects a physical memory page to evict and loads the missing page from the swap memory.

As in Section 2 we denote by $d.b$ the maximal byte address accessible by the virtual machine. We use a simulation relation $B(vm, d)$ to indicate that a (user mode) physical machine configuration $d$ encodes virtual machine configuration $vm$. Essentially, $B(vm, d)$ is the conjunction of the following two conditions:

1. For each of the $d.b/(4K)$ pages of virtual memory there is a page table entry in the physical machine, i.e. $d.b/(4K) = d.ptl$.
2. The content of virtual memory addressed by $va$ is stored in the physical memory at address $pma(d, va)$ if the corresponding valid bit is on; otherwise it is stored in the swap memory:

$$vm.m(va) = \begin{cases} d.m(pma(d, va)) & v(d, va) \\ d.sm(sma(d, va)) & \text{otherwise} \end{cases}$$

Thus the physical memory serves as a write back cache for the swap memory.

The simulation theorem for a single virtual machine has the following form:

**Theorem 3** *For all computations of the virtual machine* $(vm^0, vm^1, \dots)$ *there is a computation of the physical machine* $(d^0, d^1, \dots)$ *and there are step numbers for the physical machine* $(s(0), s(1), \dots)$ *such that for all* $i$ *we have* $B(vm^i, d^{s(i)})$.

Thus step $i$ of the virtual machine is simulated after step $s(i)$ of the physical machine. Even for simple handlers, the proof is not completely obvious since a single user mode instruction can cause two page faults. To avoid deadlock and guarantee forward progress, the page fault handler must not swap out the page that was swapped in during the last execution of the page fault handler. For details see [Hil05].

### 5.3. Synchronization Conditions

If the hardware implementation of a physical machine is pipelined or if instructions are executed out of order then an instruction $I(d^i)$ that is in the memory stage may modify a later instruction $I(d^j)$ for $j > i$ after it has been fetched. This situation is called a read

after write (RAW) hazard. $I(d^i)$ may (i) overwrite the instruction itself, (ii) overwrite its page table entry, or (iii) change the mode.

On a RAW hazard instruction fetch (in particular translated fetch implemented by a memory management unit) would not work correctly. Of course it is possible to detect such data dependencies in hardware and to roll back the computation if necessary. Alternatively, the software to be run on the processor must adhere to certain *software synchronization conventions*. Let $iaddr(d^j)$ denote the address of instruction $I(d^j)$, possibly translated. If $I(d^i)$ writes to address $iaddr(d^j)$, then an intermediate instruction $I(d^k)$ for $i < k < j$ must drain the pipe. The same must hold if $d^j$ is in user mode and $I(d^i)$ writes to $ptea(d^j, d^j.dpc)$. Finally, mode can only be changed to user mode by an `rfe` (return from exception) instruction (and the hardware guarantees that `rfe` instructions drain the pipe).

These conditions are hypotheses in the hardware correctness theorem in [DHP05]. It is easy to show that they hold for the kernels constructed later on in Section 9.

## 6. Compilation

### 6.1. C0 Semantics

In this section we summarize the results from [LPP05]. Recall that C0 is roughly speaking PASCAL with C syntax. Eventually we want to consider several programs running under an operating system. The computations of these programs are interleaved. Therefore our compiler correctness statement is based on a small steps / structured operational semantics [NN99,Win93].

In C0 types are elementary ($bool$, $int$, $\ldots$), pointer types, or aggregate ($array$ or $struct$). A type is called simple if it is an elementary type or a pointer type. We define the (abstract) size of types for simple types $t$ by $size(t) = 1$, for arrays by $size(t[n]) = n \cdot size(t)$, and for structures by $size(struct\{n_1 : t_1, \ldots, n_s : t_s\}) = \sum_i size(t_i)$. Values of variables with simple type are called *simple values*. Variables of aggregate type have *aggregate values*, which are represented as a flat sequence of simple values.

### 6.2. C0 Machine Configuration

A C0 machine configuration $c$ has the following components:

- The *program rest* $c.pr$ is the sequence of C0 statements to be executed. In [NN99] the program rest is called *code component* of the configuration.
- The current *recursion depth* $c.rd$.
- The *local memory stack* $c.lms$. It maps numbers $i \leq c.rd$ to memory frames (defined below). The global memory is $c.lms(0)$. We denote the top local memory frame of a configuration $c$ by $top(c) = c.lms(c.rd)$.
- A *heap memory* $c.hm$. This is also a memory frame.

Parameters of the configuration that do not change during a computation are

- The *type table* $c.tt$ containing information about types used in the program.

- The *function table* $c.ft$ containing information about the functions of a program. It maps function names $f$ to pairs $c.ft(f) = (c.ft(f).ty, c.ft(f).body)$ where $c.ft(f).ty$ specifies the types of the arguments, the local variables, and the result of the function, whereas $c.ft(f).body$ specifies the function body.

We are using a relatively explicit, low level memory model in the style of [Nor98]. Memory frames $m$ have the following components:

- The number $m.n$ of variables in $m$ (for local memory frames this also includes the parameters of the corresponding function definition).
- A function $m.name$ mapping variable numbers $i \in [0 : m.n - 1]$ to their names (not used for variables on the heap).
- A function $m.ty$ mapping variable numbers to their type. This permits to define the size of a memory frame $msize(m)$ as the number of simple values stored in it, namely: $msize(m) = \sum_{i=0}^{m.n-1} size(m.ty(i))$.
- A content function $m.ct$ mapping indices $0 \leq i < msize(m)$ to simple values.

A *variable* $v$ of configuration $c$ is a pair $v = (m, i)$ where $m$ is a memory frame of $c$ and $i < m.n$ is the number of the variable in the frame. The type of a variable $(m, i)$ is defined by $ty((m, i)) = m.ty(i)$.

Subvariables $S = (m, i)s$ are formed from variables $(m, i)$ by appending a selector $s = (s_1, \ldots, s_t)$, where each component of a selector has the form $s_i = [j]$ for selecting array element number $j$ or the form $s_i = .n$ for selecting the struct component with name $n$. If the selector $s$ is consistent with the type of $(m, i)$, then $S = (m, i)s$ is a *subvariable* of $(m, i)$. Selectors are allowed to be empty.

In C0, pointers $p$ may point to subvariables $(m, i)s$ in the global memory or on the heap. The value of such pointers simply has the form $(m, i)s$. Component $m.ct$ stores the current values $va(c, (m, i)s)$ of the simple subvariables $(m, i)s$ in canonical order. Values of aggregate variables $x$ are represented in $m.ct$ in the obvious way by sequences of simple values starting from the abstract base address $ba(x)$ of variable $x$.

With the help of visibility rules and bindings we easily extend the definition of $va$, $ty$, and $ba$ from variables and subvariables to expressions $e$.

### 6.3. C0 Machine Computation

Due to space restrictions we cannot give the full definition of the (small-step) transition function $\delta_C$ mapping C0 configurations $c$ to their successor configuration:

$$c' = \delta_C(c)$$

As an example we give a partial definition of the function call semantics.

Assume the program rest in configuration $c$ begins with a call of function $f$ with parameters $e_1, \ldots, e_n$ assigning the function's result to variable $v$, formally $c.pr = (v = f(e_1, \ldots, e_n); r)$. In the new program rest, the call statement is replaced by the body of function $f$ taken from the function table, $c'.pr = (c.ft(f).body; r)$ and the recursion depth is incremented $c'.rd = c.rd + 1$. Furthermore, the values of all parameters $e_i$ are stored in the new top local memory frame $top(c')$ by updating its content function at the corresponding positions: $top(c').ct_{size(ty(c,e_i))}(ba(c, e_i)) = va(c, e_i)$.

### *6.4. Compiler Correctness Theorem*

The compiler correctness statement (for programs to be run on physical or virtual machines) depends on a simulation relation $consis(aba)(c,d)$ between configurations $c$ of C0 machines and configurations $d$ of ISA machines that run the compiled program. The relation is parameterized by a function $aba$ mapping subvariables $S$ of the C0 machine to their allocated base addresses $aba(c,S)$ in the ISA machine. The allocation function may change during a computation (i) if the recursion depth and thus the set of local variables change due to calls and returns or (ii) if reachable variables are moved on the heap during garbage collection (not yet implemented).

Notice however, that in the first case only the range of the allocation function is changed: For C0 configurations $c$ and local or (sub) global variables $x$ the allocated base address $aba(x,c)$ depends only on $c$.

The simulation relation consists essentially of five conditions:

1. Value consistency $v-consis(aba)(c,d)$: This condition states that reachable elementary subvariables $x$ have the same value in the C0 machine and in the ISA machine. Let $asize(x)$ be the number of bytes needed to store a value of type $ty(x)$. Then we require $d.m_{asize(x)}(aba(c,x)) = va(c,x)$.
2. Pointer consistency $p-consis(aba)(c,d)$: This predicate requires for reachable pointer variables $p$ pointing to a subvariable $y$ that the value stored at the allocated address of variable $p$ in the ISA machine is the allocated base address of $y$, i.e. $d.m_4(aba(c,p)) = aba(c,y)$. This induces a subgraph isomorphism between the reachable portions of the heaps of the C0 and the ISA machine.
3. Control consistency $c-consis(c,d)$: This condition states that the delayed PC of the physical machine (used to fetch instructions) points to the start of the translated code of the program rest $c.pr$ of the C0 machine. We denote by $head(r)$ the first statement of statement sequence $r$ and we denote by $caddr(s)$ the address of the first assembler instruction that is generated for statement $s$. We require $d.dpc = caddr(head(c.pr))$ and $d.pc = d.dpc + 4$.[7]
4. Code consistency $code-consis(c,d)$: This condition requires that the compiled code of the C0 program be stored in the physical machine $d$ beginning at the code start address $cstart$. Thus it requires that the compiled code be not changed during the computation of the physical machine. We thereby forbid self modifying code.
5. Stack consistency $s-consis(c,d)$: this is a technical condition about stack pointers, heap pointers etc. which does not play an important role here.

**Theorem 4** *For every C0 machine computation* $(c^0, c^1, \dots)$ *there is a computation of the physical machine* $(d^0, d^1, \dots)$*, step numbers* $(s(0), s(1), \dots)$*, and a sequence of allocation functions* $(aba^0, aba^1, \dots)$ *such that for all steps* $i$ *the C0 machine and the physical machine are consistent* $consis(aba^i)(c^i, d^{s(i)})$.

A formal proof of this statement for a non optimizing compiler specified in Isabelle-HOL [NPW02] (roughly speaking: In ML) is completed and will be reported in [Lei06]. There is an implementation of the same compilation algorithm written in C0. A formal proof that the C0 implementation simulates the ML implementation is also completed

[7] For optimizing compilers this condition has in general to be weakened.

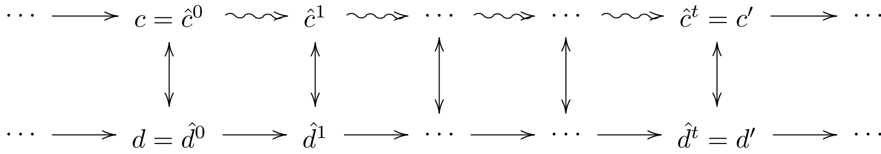


**Figure 8.** Execution of Inline Assembler Code

and will be reported in [Pet06]. In order to solve the bootstrap problem [VERa] the C0 version of the compiler was translated by an existing compiler into DLX code. That the target DLX code simulates the source code will be shown using translation validation. This is work in progress.

## 7. Inline Assembler Code Semantics

Recall that processor registers, I/O ports and user processes are not visible in the C variables of an operating system kernel written in C. Hence we must necessarily permit in our language sequences $u$ of inline assembler instructions (we do not distinguish here between assembler and machine language). We extend C0 by statements of the form $asm(u)$ and call the resulting language $\mathrm{C0_A}$. In $\mathrm{C0_A}$ the use of inline assembler code is restricted: (i) only a certain subset of DLX instructions is allowed (e.g. no load or store of bytes or half words, only *relative* jumps), (ii) the target address of store word instructions must be outside the code and data regions of the $\mathrm{C0_A}$ program or it must be equal to the allocated base address of a subvariable of the $\mathrm{C0_A}$ program with type $int$ or $unsigned\ int$ (this implies that inline assembler code cannot change the stack layout of the $\mathrm{C0_A}$ program), (iii) certain registers (e.g. the stack pointer) must not be changed, (iv) the last assembler instruction in $u$ must not be a jump or branch instruction, (v) the execution of $u$ must terminate, (vi) the target of jump and branch instructions must not be outside the code of $u$, and (vii) the execution of $u$ must not generate misalignment or illegal instruction interrupts.

In order to argue about the correctness of $\mathrm{C0_A}$ programs we must define the semantics of the newly introduced statement. A store word instruction of inline assembler code can overwrite a C variable $x$, for instance when a processor register is stored into a process control block. Hence we have to specify the effect of that store instruction on the value of $x$ in the C0 configuration. This is easily done with the help of the allocated base address functions $aba$ of the previous section (and impossible without them).

Thus consider a $\mathrm{C0_A}$ configuration $c$ with program rest $c.pr = asm(u); r$. When we enter the inline assembler portion, then the entire physical machine configuration $d$ becomes visible. In this situation we make $d$ an input parameter for the $\mathrm{C0_A}$ transition function $\delta_{C0_A}$. As pointed out above, another necessary parameter is an allocated base address function $aba$. Finally the inline assembler code will also produce a new DLX configuration $d'$. Thus we will define $(c', d') = \delta_{C0_A}(aba)(c, d)$. In all situations where we apply this definition we will have $consis(aba)(c, d)$.

The execution of $u$ leads to a physical machine computation $(d = \hat{d}^0, \ldots, \hat{d}^t = d')$ with $\hat{d}^t.dpc = caddr(head(r))$ and $\hat{d}^t.pc = \hat{d}^t.dpc + 4$ by the restrictions on inline assembler. We construct a corresponding sequence $(\hat{c}^0, \ldots, \hat{c}^t)$ of intermediate C0 machine configurations reflecting successively the possible updates of the C0 variables by

the assembler instructions (see Figure 8). We set $\hat{c}^0 = c$ except for deleting the inline assembler portion $asm(u)$ from the program rest: $\hat{c}^0.pr = r$. Let $j < t$. If predicate $sw(\hat{d}^j)$ holds, the instruction executed in configuration $\hat{d}^j$ writes the value $v = \hat{d}^j.gpr(RD(\hat{d}^j))$ to the word at address $ea(\hat{d}^j)$ as defined in Section 2. If this effective address is equal to the allocated base address of a C0 variable $x$, then we update the corresponding variable in configuration $\hat{c}^{j+1}$ such that $va(\hat{c}^{j+1}, x) = v$:

$$sw(\hat{d}^j) \wedge (ea(\hat{d}^j) = aba(c, x) \Rightarrow va(\hat{c}^{j+1}, x) = \hat{d}^j.gpr(RD(\hat{d}^j))$$

Finally the result of the $C0_A$ transition function is defined by $c' = \hat{c}^t$ and $d' = \hat{d}^t$. This definition keeps configurations consistent:

**Lemma 1** *If the program rest of c starts with an inline assembler statement we have:*

$$consis(aba)(c, d) \Rightarrow consis(aba)(\delta_{C0_A}(aba)(c', d'))$$

## 8. Communicating Virtual Machines (CVM)

### *8.1. CVM Semantics*

We now introduce communicating virtual machines (CVM), a model of computation for a generic operating system kernel interacting with a fixed number of user processes. While the CVM is running, the kernel can only be interrupted by reset. Kernels with this property are called non-preemptive[8]. CVM uses the C0 language semantics to model computations of the (abstract) kernel and virtual machines to model computations of user processes. It is a pseudo-parallel model in the sense that in every step of computation either the kernel or one user process can make progress.

From a kernel implementor's point of view, CVM encapsulates the low-level functionality of a microkernel and provides access to it as a library of functions, the so-called CVM primitives. Accordingly, the abstract kernel may be 'linked' with the implementation of these primitives to produce the concrete kernel, a $C0_A$ program, that may be run on the target machine. This construction and its correctness will be treated in Section 9.

In the following sections we define CVM configurations, CVM computations, and show how abstract kernels implement system calls as regular C0 function calls.

### *8.2. CVM Configuration*

A CVM configuration $cvm$ has the following components:

- User processes are modeled by virtual machine configurations $cvm.vm(u)$ having indices $u \in \{1, \ldots, P\}$ (and $P$ fixed, e.g. $P = 128$).
  - ∗ Each user process has an individual page table 'lengths' $cvm.vm(u).ptl$. The memory available to virtual machines can be de- or increased dynamically.

[8]Premptive kernels require dealing with nested interrupts. A theory of nested interrupts is outlined in Chapter 5 of [MP00].

- A C0 machine configuration $cvm.c$ represents the so-called *abstract kernel*. We require the kernel configuration, in particular its initial configuration, be in a certain form:
  - ∗ Certain functions $f \in CVMP$, the CVM primitives, must be declared only, i.e. their body must be empty. Its arguments and effects are described below.
  - ∗ In addition to the cvm primitives a special function called $kdispatch$ must be declared. It takes two integer arguments and returns an integer. An invocation of the $kdispatch$ function must eventually result in a function call of the CVM primitive $v = start(e)$, which passes control to the user processes determined by the current value $va(cvm.c, e)$ of expression $e$.
- The component $cvm.cp$ denotes the current process: $cvm.cp = 0$ means that the kernel is running while $cvm.cp = u > 0$ means that user process $u$ is running.
- The $cvm.f$ component denotes the state of one external device[9] capable of interrupting user processes with an ISA interrupt signal $eev$.

### 8.3. CVM Computation

In every step of a CVM computation a new CVM configuration is computed from an old configuration $cvm$, an oracle input $eev$, and from a device specific external input $fdin$:

$$cvm' = \delta_{CVM}(cvm, eev, fdin)$$

The external input $fdin$ only affects the device state $cvm'.f$. Updates of this state are device specific and are not treated here.

*User computation.* If the current process $u = cvm.cp$ in configuration $cvm$ is non-zero then user process $vm(u)$ does a step:

$$cvm'.vm(u) = \delta_D(cvm.vm(u))$$

If no interrupt occurred, i.e. $\neg JISR(cvm.vm(u), eev)$ then user process $vm(u)$ keeps running:

$$cvm'.cp = u$$

Otherwise execution of the abstract kernel starts. Recall from Section 2.3 on interrupt semantics, that in case of an interrupt the masked cause register is saved into the exception cause register $eca$ and that certain data necessary for handling the exception is stored in register $edata$. The kernels entry point is the function $kdispatch$ that is called with the saved exception cause register $cvm.vm(u).eca$ and the saved exception data register $cvm.vm(u).edata$ as parameters. We set the current process component and the kernel's recursion depth to zero:

$$\begin{aligned} cvm'.cp &= 0 \\ cvm'.c.rd &= 0 \\ cvm'.c.pr &= (v = kdispatch(cvm.vm(u).eca, cvm.vm(u).edata)) \end{aligned}$$

[9] Dealing with more devices is not necessary here; it is not much more difficult.

*Kernel computation.* Initially (after power-up) and after an interrupt, as seen above, the kernel starts execution with a call of the function $kdispatch$. User process execution continues when the kernel calls the CVM primitive $start$.

If we have $cvm.cp = 0$ and the kernel's program rest does not start with a call to a CVM primitive, a regular C0 semantics step is performed:

$$cvm'.c = \delta_C(cvm.c)$$

Otherwise, we have $cvm.cp = 0$ and $cvm.c.pr = (v = f(e_1, \ldots, e_n); r)$ for a CVM primitive $f$, an integer variable $v$ and integer expressions $e_1$ to $e_n$. Although the implementation of the CVM primitives involves inline assembler code, their semantics can be specified in the pseudo parallel CVM model by their effect on the user processes $vm(u)$ and on the device $f$.

Below we describe a few selected CVM primitives. We ignore any preconditions or border cases; these are straightforward to specify and resolve:

- The $start(e)$ primitive hands control over to the user process specified by the current value of expression $e$:

  $$cvm'.cp = va(cvm.c, e)$$

  By this definition, the kernel stops execution and is restarted again on the next interrupt (with a fresh program rest as described before).
- The $alloc(u, x)$ primitive increases the memory size of process $U = va(cvm.c, u)$ by $X = va(cvm.c, x)$ pages:

  $$cvm'.vm(U).ptl = cvm.vm(U).ptl + X$$

  The new pages are cleared:

  $$\forall y \in [cvm.vm(U).ptl : cvm.vm(U).ptl + 4K - 1] : cvm'.vm(U).m(y) = 0^8$$

- The primitive $free(u, x)$ that frees $X = va(cvm.c, x)$ pages of user process $U = va(cvm.c, u)$ is defined in a similar way.
- The primitive $copy(u_1, a_1, u_2, a_2, d)$ copies a memory region between user processes $U_1 = va(cvm.c, u_1)$ and $U_2 = va(cvm.c, u_2)$. The start addresses in the memory of the source process $U_1$ and the destination process $U_2$ are given by $A_1 = va(cvm.c, a_1)$ and $A_2 = va(cvm.c, a_2)$ respectively. The number of bytes to be copied is given by $D = va(cvm.c, d)$:

  $$cvm'.vm(U_2).m_D(A_2) = cvm.vm(U_1).m_D(A_1)$$

- Primitives copying data between user processes and I/O ports and between C variables of the kernel and I/O ports are defined in a similar way.

- The primitive $e = getgpr(r, u)$ reads general purpose register $R = va(cvm.c, r)$ of user process $U = va(cvm.c, u)$ and assigns it to the (sub)variable specified by expression $e$:

$$va(cvm'.c, e) = cvm.vm(U).gpr(R)$$

As described below, this primitive is used to read parameters of system calls.
- The primitive $setgpr(r, u, e)$ writes the current value of expression $e$ into general purpose register $R$ of process $U$:

$$cvm'.vm(U).gpr(R) = va(cvm.c, e)$$

This primitive is used to set return values of system calls.

### *8.4. Binary Interface of Kernels*

Before we deal with the implementation of CVM and a proof for its correctness, we show how to build a kernel by appropriately specializing the generic abstract kernel of CVM.

The obvious means for a user process to invoke a system call is to use the trap instruction that causes an internal interrupt. If the kernel provides $k$ trap handlers, then the user can specify the handler to be invoked using the immediate constant $i$ being part of the trap instruction, where $i \in [0 : k - 1]$. A so called kernel call definition function $kcd$ maps immediate constants $i \in [0 : k - 1]$ to names of functions declared in the abstract kernel. Thus $kcd(i)$ is simply the name of the C function (including CVM primitives) handling a trap with immediate constant $i$. For each $i$, let $np(i) < 20$ be the number of parameters[10] of function $kcd(i)$. We require that user processes pass the parameters for function $kcd(i)$ in general purpose registers $gpr[1 : np(i)]$. Together with the specification of the functions $kcd(i)$ this is the entire binary interface definition.

Implementation by specialization of the abstract CVM kernel is completely straight forward. First of all the kernel maintains a variable *cup* keeping track of the user process that is currently running or that has been running before the kernel started execution:

$$cvm.cp > 0 \Rightarrow va(cvm.c, cup) = cvm.cp$$

Assume $cvm.cp = u > 0$ and user $vm(u)$ executes the trap instruction with immediate constant $i$. Furthermore assume that the trap instruction activates internal event line $iev(5)$, as described in Section 3.5, and that no interrupts with higher priority (lower index) are active simultaneously. Then the masked cause vector $0^{26}10^5$ is saved into the exception cause register $eca[31 : 0]$ and parameter $i$ is saved into the exception data register $edata$:

$$cvm'.vm(u).eca = 0^{26}10^5$$
$$cvm'.vm(u).edata = i$$

According to the CVM semantics the abstract kernel starts running with the function call $kdispatch(eca, edata)$ where $eca = 0^{26}10^5$ and $edata = i$. By a case split

[10] Assume for simplicity they are of type integer.

on $eca$ the handler concludes that a trap instruction needs to be handled. Hence the handler invokes the function call $f(e_1, \ldots, e_{np(i)})$, where $f = kcd(i)$ using the parameters computed by the assignment $e_i = getgpr(i, cup)$.

Let $cvmd$ be the CVM configuration immediately after execution of the call of $kcd$. Then we easily derive from the semantics of CVM and C0:

**Lemma 2 (Intended Handler Called with the Intended Parameters)**

$$\begin{aligned} cvmd.rd &= cvm'.rd + 1 = 1 \\ cvmd.c.pr &= cvm.c.ft(f).body; r \qquad \textit{for some } r \\ top(cvmd).ct(j) &= cvm.vm(u).gpr(j) \qquad \textit{for all } j \in [1 : np(i)] \end{aligned}$$

This lemma formalizes the idea that an interrupt is something like a function call of the handler. Comparing with the C0 semantics in Section 6.1 we see that the trap instruction indeed formally causes a function call of the handler. The function call is however remote, because it is executed by a process (the abstract kernel) different from the calling process (virtual machine $vm(u)$).

## 9. CVM Implementation and Correctness

### *9.1. Concrete Kernel and Linking*

So far we have talked about the abstract kernel, but we have argued mathematically only about its configurations $c$. Now we also argue about its source code that we denote by $sak$. We describe how to obtain the source code $sck$ of the so called *concrete kernel* by linking $sak$ with the source code of some CVM implementation $scvm$ using some link operator $ld$:

$$sck = ld(sak, scvm)$$

Note that $sak$ is a pure C0 program, whereas $scvm$ and $sck$ are $C0_A$ programs. The function table of the linked program $sck$ is constructed from the function tables of the input programs. For functions present in both programs, *defined functions* (with a non-empty body) take precedence over *declared functions* (without a body). We do not formally define the $ld$ operator here; it may only be applied under various restrictions concerning the input programs, e.g. the names of global variables of both programs must be distinct, function signatures must match, and no function may be defined in both input programs.

We require that the abstract kernel $sak$ defines $kdispatch$ and declares all CVM primitives while the CVM implementation $scvm$ defines the primitives and declares $kdispatch$.

In analogy to the $consis$ relation of the compiler correctness proof we define a relation $kconsis(kalloc)(c, cc)$ stating that abstract kernel configuration $c$ is coded by concrete kernel configuration $cc$. The configuration $cc$ is a tuple consisting of a C0 machine configuration $cc.c$ and a physical machine configuration $cc.d$.

The function $kalloc$ maps subvariables $x$ of abstract kernel configuration $c$ to subvariables $kalloc(x)$ of concrete kernel configuration $cc$.

Linking is less complex than compiling. The definition of the $kconsis$ relation has only three parts:

1. $e - kconsis(kalloc)(c, cc)$: All reachable elementary (sub) variables $x$ of the abstract kernel configuration $c$ and the values of $x$ in the concrete kernel coincide:

$$va(c, x) = va(cc.c, x)$$

2. $kalloc$ is a graph isomorphism between reachable portions of the heaps. For all reachable pointer variables $p$ of abstract kernel configuration $c$, pointing to subvariable $v$, the following holds:

$$(va(c, p) = v) \Rightarrow va(cc.c, kalloc(p)) = kalloc(v)$$

3. $c - kconsis$: The program rest of the concrete kernel is a prefix of the program rest of the abstract kernel. For technical reason there is a particular suffix $r$ containing 'dangling returns'. This suffix is cleared when the kernel is started the next time (see Section 8.3).

$$cc.c.pr = c.pr; r$$

*9.2. Data Structures*

The CVM implementation maintains data structures for the simulation of the virtual machines, i.e. for the support of multiprocessing. These include:

1. An array of process control blocks $pcb[u]$ for the kernel ($u = 0$) and the user processes ($u > 0$). Process control blocks are structs with components $pcb[u].R$ for every processor register $R$ of the *physical* machine.
2. A single integer array $ptarray$ on the heap holds the page tables of all user processes in the order of the process numbers $u$. The function $ptbase(u)$ defines the start index of the page table for process $u$:

$$ptbase(u) = \sum_{j<u} (pcb[j].ptl + 1)$$

Since the C array $ptarry$ is indexed by words and not by bytes we define the page table entry for virtual address $va$ and process $u$ as:

$$pte(u, va) = ptarray[ptbase(u) + va.px]$$

Notice that we have faked pointer arithmetic on the page table array, but formally we just barely managed to dance around it. The physical page address and valid bit are defined by C expressions.

$$pma(u, va) = pte(u, va)[31 : 12] \circ va.bx$$
$$v(u, va) = pte(u, va)[11]$$

Swap memory addresses $sma(u, va)$ are computed by C function in an analogous way. We require that the compiler computes the allocated base address of array $ptarray$ as a multiple of the page size $4K$.

3. Data structures (in the simplest case doubly-linked lists) for the management of physical and swap memory (including victim selection for page faults).
4. The variable $cup$ keeping track of the current user process thus encoding the $cvm.cp$ component (unless the kernel is running).

### 9.3. Entering System Mode after an Interrupt

When the mode bit in the concrete kernel flips from user to system mode, the program rest is initialized with $init_1; init_2$. In all cases except reset, the first part $init_1$ will (i) write all processor registers $R$ to the process control block $pcb[cup].R$ of the process $cup$ that was interrupted while it was running and (ii) restore the registers of the kernel from process control block $pcb[0]$.

In the second part $init_2$, the CVM implementation detects whether the interrupt was due to a page fault or to other causes. Page faults are handled silently without calling the abstract kernel (cf. below). For other interrupts, we call $kdispatch$ with the parameters already obtained from the C variables $pcb[cup]$:

$$kdispatch(pcb[cup].eca, pcb[cup].edata)$$

### 9.4. Leaving System Mode

A call of $start(cup)$ will switch to user mode again. It is implemented using inline assembler. We write the physical processor registers to $pcb[0]$ in order to save the concrete kernel state. Then we restore the physical processor registers for process $cup$ from $pcb[cup]$ and execute an `rfe` instruction (return from exception).

### 9.5. Page Fault Handler

The page fault handler maintains a simulation relation $B$ as described in Section 5.2. With correct page fault handlers, user mode steps in the physical machine without interrupts simulate steps of a virtual machine. Note that a single user mode instruction can produce up to two page faults: One during instruction fetch and one during a load or store operation. In order to prevent even more page faults the page most recently swapped in must not be choose as the victim page to be swapped out (as it is possible with pure random selection of the victim page).

To reason about multiple user processes $u$, we have to slightly modify and extend the $B$ relation. Let $u$ be an index of a user process/virtual machine. Let $cvm$ be a CVM configuration and let $cc$ be a configuration of the concrete kernel. We define predicate $B(u)(cvm, cc)$ stating that the configuration $cvm.vm(u)$ of user process $u$ is coded by configuration $cc$.

1. Processor registers of $vm(u)$ are stored in the physical processor registers, if process $u$ is running; otherwise they are stored in the process control blocks:

$$cvm.vm(u).gpr(r) = \begin{cases} cc.d.gpr(r) & cvm.cp = u \\ va(cc.c, pcb[u].gpr(r)) & \text{otherwise} \end{cases}$$

2. The memory content $vm(u)(a)$ is stored in the physical memory at the corresponding physical memory address, if the valid bit of virtual address $a$ is 1, otherwise it is stored in swap memory at the swap memory address:

$$cvm.vm(u).m(a) = \begin{cases} cc.d.m(va(cc.c, pma(u,a))) & va(cc.c, v(u,a)) = 1 \\ cc.d.sm(va(cc.c, sma(u,a))) & \text{otherwise} \end{cases}$$

### 9.6. Implementation of the CVM Primitives

The implementation of CVM primitives like $e = getgpr(u, r)$ and $e = setgpr(u, r, g)$ is straightforward. Let $U = va(cc.c, u)$, $R = va(cc.c, r)$, $G = va(cc.c, g)$ and $E = va(cc.c, e)$, then:

$$E = va(cc.c, pcb[U].gpr(R))$$
$$va(cc.c, pcb[U].gpr(R)) = G$$

For the CVM primitives *alloc* and *free* the page table length of the process has to be increased or decreased and –we have chosen a very simple implementation– lots of page table entries in $ptarry$ above the portion of the modified user process have to me moved around in the page table array. Various other data structures concerning memory management have to be adjusted as well. Such operations are closely interconnected with the page fault handler. Since the page tables are accessible as a C0 data structure, inline assembler is only required to clear newly allocated physical pages. Similarly, the implementation of the *copy* primitive requires assembler code to copy pages of physical memory between user processes.

### 9.7. CVM Correctness Theorem

The correctness proof of the cvm deals simultaneously with computations in three computational models:

1. The CVM consisting of a C0 machine and several virtual machines; configurations are denoted by $cvm$.
2. An intermediate model for the $C0_A$ computation of the concrete kernel; configurations are denoted by $cc.c$
3. The physical machine model; configurations are denoted by $cc.d$

**Theorem 5** *Consider an input sequence of external interrupts ($eev^0, eev^1, \ldots$) and a CVM computation ($cvm^0, cvm^1, \ldots$) defined with this input sequence. Then there exists (i) a concrete kernel computation ($cc.c^0, cc.c^1, \ldots$), (ii) a physical machine computation ($cc.d^0, cc.d^1, \ldots$), (iii) two sequences of allocation functions ($aba^0, aba^1, \ldots$) and ($kalloc^0, kalloc^1, \ldots$) and finally (iv) two sequences of step numbers ($s^0, s^1, \ldots$) and ($t^0, t^1, \ldots$) such that:*

1. *The abstract kernel component $cvm^i.c$ of the CVM computation after $i$ steps is coded by the concrete kernel configuration $cc^{s(i)}$ after $s(i)$ steps:*

$$kconsis(kalloc^i)(cvm^i.c, cc^{s(i)})$$

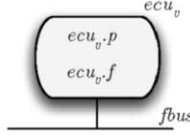



**Figure 9.** Electronic Control Units

2. *The concrete kernel configuration $cc.c^{s(i)}$ after $s(i)$ steps is coded by configuration $cc.d^{t(i)}$ of the physical machine after $t(i)$ instructions. Recall that on the physical machine the compiled concrete kernel is executed:*

$$consis(aba^i)(cc.c^{s(i)}, cc.d^{t(i)})$$

3. *The user machines $cvm^i.vm(u)$ after $i$ steps of the CVM computation are coded by the configuration $cc^{t(i)}$ of the concrete kernel after $t(i)$ instructions:*

$$B(u)(cvm^i, cc^{t(i)})$$

The correctness theorem is proven by induction over the steps $i$ of the CVM computations. Depending on the current process number $cvm^i.cp$ and the interrupts occurring, the proof uses compiler correctness (see Section 6.4), the correctness of memory management mechanisms (see Section 5.2), and detailed arguments about inline assembler code using $C0_A$ semantics (see Section 7).

## 10. Parallel Hardware Overview

So far we only have considered systems with a single processor and a device. In what follows we construct particular hardware devices serving as interfaces to a FlexRay-like bus called $fbus$. The devices will be called FlexRay-like interfaces or short f-interfaces. A processor together with a device will be called an electronic control unit (ECU).

We will consider $p$ electronic control units $ecu_v$, where $v \in \{0, \ldots, p-1\}$, which are communicating over a common $fbus$. At the ISA level, an ECU configuration $ecu_v = (ecu_v.d, ecu_v.f)$ is a pair consisting of a processor configuration $ecu_v.d$, and a configuration $ecu_v.f$ of an f-interface, see Figure 9.

From an interface configuration $ecu_v.f$ we define two user-visible buffers: A send buffer $sb(ecu_v)$ and a receive buffer $rb(ecu_v)$. Each buffer is capable of holding a message of $\ell$ bytes.

In the distributed system all communications and computations proceed in rounds $r$ where $r \in \mathbb{N}$. As depicted in Figure 10 each round is divided into an (even) number of slots $s$ where $s \in \{0, \ldots, ns-1\}$. The tuple $(r, s)$ refers to slot $s$ in round $r$. On each ECU, boundaries between slots will be determined by local timer interrupts every $T$ hardware cycles. At the beginning of each round the local timers are synchronized.

Given a slot $(r, s)$ we define the predecessor $(r, s)-1$ and successor $(r, s)+1$ according to the lexicographical order of slots. We denote by $d_v(r, s)$ the first and by $e_v(r, s)$ the last ISA configuration of $ecu_v$ during slot $(r, s)$.

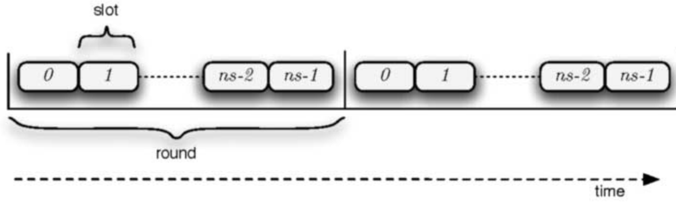



**Figure 10.** Slots and Rounds

ECUs of the system communicate according to a fixed schedule that is identical for each round. The $send$ function specifies for all rounds $r$ the electronic control unit ECU that owns the bus during slot $(r, s)$:

$$send : \{0, \ldots, ns - 1\} \to \{0, \ldots, p - 1\}$$

During slot $(r, s)$ the content of the send buffer of $ecu_{send(s)}$ at the end of the previous round $(r, s) - 1$ is broadcast to the receive buffers of all units $ecu_u$ and becomes visible there at the beginning of the next round $(r, s) + 1$:

$$\forall u, r, s : sb(e_{send(s)}((r, s) - 1)) = rb(d_u((r, s) + 1)) \quad (1)$$

In Sections 11 to 14 we will outline the proof of a hardware correctness theorem for the entire distributed system justifying this programming model. This theorem establishes for each $ecu_v$ at the start of each slot $(r, s)$ the naive simulation relation $sim$ from Section 3.4 between the ISA configuration $d_v(r, s)$ before the execution of the first instruction of the slot and the corresponding hardware configuration $h_v(r, s)$ during the first hardware cycle of the slot:

$$sim(d_v(r, s), h_v(r, s))$$

## 11. Serial Interface

The hardware of each ECU is clocked by an oscillator with a nominal clock period of say $\tau_{ref}$. For all $v$ the individual clock periods $\tau_v$ of $ECU_v$ are allowed to deviate from the nominal period by $\delta = 0.15\%$:

$$\mid \tau_v - \tau_{ref} \mid \leq \tau_{ref} \cdot \delta$$

This limitation can be easily achieved by current technology.

With $\Delta = 2\delta/(1 - \delta)$ we easily bound for all $u$ and $v$ the relative deviation of individual clock periods among each other by:

$$\mid \tau_v - \tau_u \mid \leq \tau_v \cdot \Delta$$

Consider a situation, where a sending ECU puts data on the bus and these data are sampled into registers of receiving ECUs. Then, due to the clock drift, we cannot

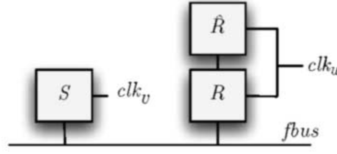



**Figure 11.** Serial Interface

guarantee that the set up and hold times of the receiving registers are obeyed at all clock edges. This problem occurs whenever computers without a common clock exchange data. It is solved by serial interfaces using a nontrivial protocol. Therefore we first need a hardware correctness proof of a serial interface as prescribed by the FlexRay standard.

### *11.1. Hardware Model with Continuous Time*

The problems solved by serial interfaces can by their very nature not be treated in the standard digital hardware model with a single digital clock $clk$. Nevertheless, we can describe each $ECU_v$ in a standard digital hardware model having its own hardware configuration $h_v$.

In order to argue about a sender register $S$ of a sending ECU that is transmitting data via the $fbus$ to a receiver register $R$ of a receiving ECU, as depicted in Figure 11, we have to extend the digital model.

For the registers –and only for the registers– connected to the $fbus$ we extend the hardware model such that we can deal with the concepts of propagation delay ($tpd$), set-up time ($ts$), hold time ($th$) and metastability of registers from hardware data sheets. In the extended model used near the $fbus$ we therefore consider time to be a real valued variable $t$. Given some offset $c_v < \tau_v$ the date of the clock edge $e_v(i)$ that starts cycle $i$ on $ECU_v$ is defined by:

$$e_v(i) = c_v + i \cdot \tau_v \tag{2}$$

In this continuous time model the content of the a sender register $S$ at time $t$ is denoted by $S(t)$.

We now have enough machinery to define in the continuous time model the output of a sender register $S_v$ on $ECU_v$ during cycle $i$ of $ECU_v$, i.e. for $t \in (e_v(i), e_v(i+1)]$. If in cycle $i-1$ the digital clock enable $Sce(h_v^{i-1})$ signal was off, we see during the whole cycle the old digital value $h_v^{i-1}.S$ of the register. If the update enable signal was on, then during the propagation delay $tpd$ we cannot predict what we see, which we denote by $\Omega$. When the propagation delay has passed, we see the new digital value of the register, which is equal to the digital input $Sdin(h_v^{i-1})$ during the previous cycle (see Figure 12).

$$S_v(t) = \begin{cases} h_v^{i-1}.S & \neg Sce(h_v^{i-1}) \\ \Omega & Sce(h_v^{i-1}) \wedge t \leq e_v(i) + tpd \\ Sdin(h_v^{i-1}) & Sce(h_v^{i-1}) \wedge t > e_v(i) + tpd \end{cases}$$

The $fbus$ is an open collector bus modeled for all $t$ by:

$$fbus(t) = \bigwedge_v S_v(t)$$

Now consider a receiver register $R_u$ on $ECU_u$ whose clock enable is continuously turned on; thus the register always samples from the $fbus$. In order to define the new digital value $h_u^j.R$ of register $R$ during cycle $j$ on $ECU_u$ we have to consider the value of the $fbus(t)$ in the time interval $(e_u(j)-ts, e_u(j)+th)$, i.e. from the clock edge minus the set-up time until the clock edge plus the hold time. If during that time the $fbus$ has a constant digital value $x$, the register samples that value:

$$\exists x \in \{0,1\}\ \forall t \in (e_u(j)-ts, e_u(j)+th) : fbus(t) = x \Rightarrow h_u^j.R = fbus(e_u(j))$$

Otherwise we define $h_u^j.R = \Omega$.

We have to argue how to deal with unknown values $\Omega$ as input to digital hardware. We will use the output of register $R$ only as input to a second register $\hat{R}$ whose clock enable is always turned on, too. If $\Omega$ is clocked into $\hat{R}$ we assume that $\hat{R}$ has an unknown but digital value:

$$h_u^j.R = \Omega \Rightarrow h_u^{j+1}.\hat{R} \in \{0,1\}$$

Indeed, in real systems the counterpart of register $\hat{R}$ exists. The probability that $R$ becomes metastable for an entire cycle *and* that this causes $\hat{R}$ to become metastable too is for practical purposes zero. This is exactly what has been formalized above. Note that our model uses different but fixed individual clock periods $\tau_v$.

There is no problem to extend the model to deal with jitter. Let $\tau_v(i)$ denote the length of cycle $i$ on $ECU_v$, then we require for all $v$ and $i$:

$$\tau_v(i) \in [\tau_{ref} \cdot (1-\delta), \tau_{ref} \cdot (1+\delta)]$$

The time $e_v(i)$ of the $i$-th clock edge on $ECU_j$ is then defined as:

$$e_v(i) = \begin{cases} c_v & i = 0 \\ e_v(i-1) + \tau_v(i-1) & \text{otherwise} \end{cases}$$

This does not complicate the subsequent theory significantly.

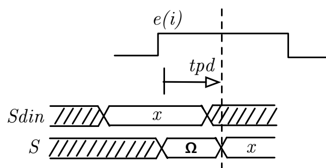



**Figure 12.** Sender Register

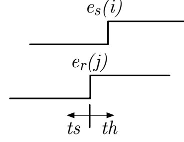



**Figure 13.** Clock Edges

### *11.2. Continuous Time Lemmas for the Bus*

Consider a pair of ECUs, where $ECU_s$ is the sender and $ECU_r$ is a receiver in a given slot. Let $i$ be a sender $cycle$ such that $Sce(h_s^{i-1}) = 1$, i.e. the output of $S$ is not guaranteed to stay constant at time $e_s(i)$. This change can only affect the value of register $R$ of $ECU_r$ in cycle $j$ if it occurs before the sampling edge $e_r(j)$ plus the hold time $th$, i.e. $e_s(i) < e_r(j) + th$. Figure 13 shows a situation where due to a hold time violation we have $e_s(i) > e_r(j)$. The first cycle that is possibly being affected is denoted by:

$$cy_{r,s}(i) = \min\{j \mid e_s(i) < e_r(j) + th\}$$

In what follows we assume that all ECUs other than the sender unit $ECU_s$ put the value 1 on the bus (hence $fbus(t) = S_s(t)$ for all $t$ under consideration). Furthermore we consider only one receiving unit $ECU_r$. Because the indices $r$ and $s$ are fixed we simply write $cy(i)$ instead of $cy_{r,s}(i)$.

There are two essential lemmas whose proof hinges on the continuous time model. The first lemma considers a situation, where we activate the clock enable $Sce$ of the sender ECU in cycle $i-1$ but not in the following seven cycles. In the digital model we then have $h_s^i.S = \ldots = h_s^{i+7}.S$ and in the continuous time model we observe $x = fbus(t) = S_v(t) = h_s^i.S$ for all $t \in [e_s(i) + tpd, e_s(i+8)]$. We claim that $x$ is correctly sampled in at least six consecutive cycles

**Lemma 3 (Correct Sampling Interval)** *Let the clock enable signal of the $S$ register be turned on in cycle $i-1$, i.e. $Sce(h_s^{i-1}) = 1$ and let the same signal be turned off in the next seven cycles, i.e. $Sce(h_s^j) = 0$ for $j \in \{i, \ldots, i+6\}$ then:*

$$h_r^{cy(i)+k}.R = h_s^i.S \quad \textit{for } k \in \{1, \ldots, 6\}$$

The second lemma simply bounds the clock drift. It essentially states that within 300 cycles clocks cannot drift by more than one cycle; this is shown using $\delta \leq 0.15\%$.

**Lemma 4 (Bounded Clock Drift)** *The clock drift for a given cycle $m \in \{1, \ldots, 300\}$ is bounded by:*

$$cy(i) + m - 1 \leq cy(i+m) \leq cy(i) + m + 1$$

Detailed proofs of very similar lemmas are to be found in [Pau05,BBG$^+$05], a formal proof is reported in [Sch06a].

### *11.3. Serial Interface Construction and Correctness*

Recall that for natural numbers $n$ and bits $y$ we denote by $y^n$ the string in which bit $y$ is replicated $n$ times, e.g. $0^4 = 0000$. For strings $x[0 : k-1]$ consisting of $k$ bits $x[i]$ we denote by $8 \cdot x$ the string obtained by repeating each bit eight times:

$$8 \cdot x = x[0]^8 \circ \cdots \circ x[k-1]^8$$

Our serial interface transmits messages $m[0 : \ell - 1]$ consisting of $\ell$ bytes $m[i]$ from a send buffer $sb$ of the sending ECU to a receive buffer $rb$ of the receiving ECU.

The following protocol is used for transmission (see Figure 14). One creates from message $m$ a frame $f(m)$ by inserting falling edges between the bytes and adding some bits at the start and the end of the frame:

$$f(m) = 0110m[0] \cdots 10m[\ell - 1]01$$

In $f(m)$ we call the first zero the transmission start sequence ($TSS$), the first one the frame start sequence ($FSS$), the last zero the frame end sequence ($FES$) and the last one the transmission end sequence ($TES$). The two bits producing a falling edge before each byte are called the byte start sequence ($BS0$, $BS1$). The sending ECU broadcasts $8 \cdot f(m)$ over the $fbus$.

Figure 15 shows a simplified view on the hardware involved in the transmission of a message. On the sender side, there is an automaton keeping track which bit of the frame is currently being transmitted. This automaton inserts the additional protocol bits around the message bytes. Hardware for sending each bit eight times and for addressing the send buffer is not shown.

On the receiver side there is the automaton from Figure 14 (the automaton on the sender side is very similar) trying to keep track of which bit of the frame is currently transmitted. That it does so successfully requires proof.

The bits sampled in register $\hat{R}$ are processed in the following way. The voted bit $v$ is computed by applying a majority vote to the last five sampled bits. These bits are given by the $\hat{R}$ register and a 4-bit shift register as depicted in Figure 16.

According to Lemma 3 for each bit of the frame a sequence of at least six bits is correctly sampled. The filtering essentially maintains this property. If the receiver succeeds to sample that sequence roughly in the middle, he wins. For this purpose the receiver has a modulo-8 counter (see Figure 17) trying to keep track of which of the eight identical copies of a frame bit is currently transmitted. When the counter value equals four a strobe bit is produced. For frame decoding the voted bit is sampled with the strobed bit. The automaton trying to keep track of the protocol is also clocked with this strobe bit.

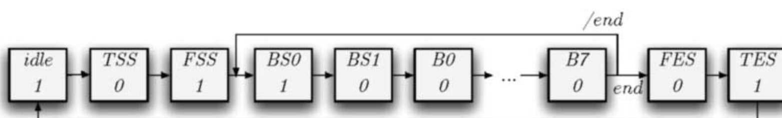



**Figure 14.** Frame Encoding

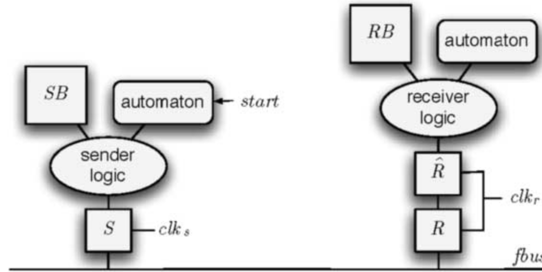



**Figure 15.** Send and Receive Buffer

Clocks are drifting, hence the hardware has to perform a low level synchronization. The counter is reset by a $sync$ signal in two situations: At the beginning of a transmission or at an expected falling edge during the byte start sequence. Abbreviating signals $s(h_r^i)$ with $s^i$ we write

$$sync^i = (idle^i \vee BS0^i) \wedge (\neg v^i \wedge v^{i-1})$$

The crucial part of the correctness proof is a lemma arguing simultaneously about three statements by induction over the receiver cycles:

**Lemma 5 (Transmitting Single Messages)** *Three hypothesis:*

1. *The state of the automaton keeps track of the transmitted frame bit.*
2. *The $sync$ signal is activated at the corresponding falling edge of the voted bit between $BS0$ and $BS1$.*
3. *Sequences of identical bit are sampled roughly in the middle.*

Formalizing this lemma (as done in [Pau05,Sch06b]) requires a detailed look on the automaton as well as on the sender- and receiver logic which both is not possible here due to space restrictions. A formal proof of such a lemma in an abstract model, which was obtained largely by automatic methods, is reported in [BP06]; a formal proof of the lemma in our hardware model is reported in [Sch06b].

We sketch the proof: Statement 1 is clearly true in the $idle$ state. From statement 1 follows that the automaton expects the falling edges of the voted signal exactly when the

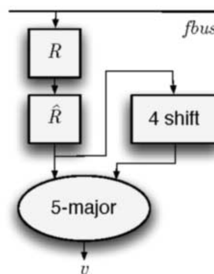



**Figure 16.** Receiver Logic

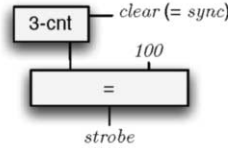



**Figure 17.** Strobe Signal

sender generates them. Thus the counter is well synchronized after these falling edges. This shows statement 2. Immediately after synchronization the receiver samples roughly in the middle. There is a synchronization roughly every 80 sender cycles. By Lemma 4 and because $80 < 300$, the sampling point can wander by at most one bit between activations of the $sync$ signal. This is good enough to stay within the correctly sampled six copies. This shows statement 3. If transmitted frame bits are correctly sampled, then the automaton keeps track of them. This shows statement 1.

Let $t_0$ be the time (not the cycle) when the $start$ signal of the sender is activated. Let $t_1$ be the time, when all automata have reached the $idle$ state again and all write accesses to the receive buffer are completed. Let

$$tc = 45 + 80 \cdot \ell$$

be the number of 'transmission cycles'. Then:

**Lemma 6 (Correct Message Transfer With Time Bound)** *All messages are correctly transmitted, and the transmission does not last longer than $tc$ sender cycles:*

$$rb(t_1) = sb(t_0)$$
$$t_1 - t_0 \leq tc \cdot \tau_s$$

Intuitively, the product $80 \cdot \ell$ in the definition of $tc$ comes from the fact that each byte produces 10 frame bits and each of these is transmitted 8 times. The four bits added at the start and the end of the frame contribute $4 \cdot 8 = 32$. The remaining 13 cycles are caused by delays in the receiver logic, in particular by the delay in the shift register before the majority voter.

## 12. FlexRay-Like Interfaces and Clock Synchronization

Using the serial interfaces from the last section we proceed in the construction of entire f-interfaces. The results from this section were first reported in [Pau05,KP06].

### *12.1. Hardware Components*

Recall that we denote hardware configurations of $ECU_v$ by $h_v$. If the index $v$ of the ECU does not matter, we drop it. The hardware configuration is split into a processor configuration $h.p$ and an interface configuration $h.f$. In addition to the registers of the serial interface, the essential components of the hardware configuration $h.f$ of our (non fault tolerant) FlexRay-like interface are:

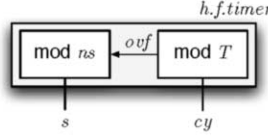



**Figure 18.** Hardware Timer

- Double buffers $h.f.sb(par)$ and $h.f.rb(par)$, where $par \in \{0, 1\}$, implementing the user-visible send and receive buffers.
- The registers of a somewhat non trivial timer $h.f.timer$.
- Configuration registers.

The construction of the hardware timer $h.f.timer$ is sketched in Figure 18. The low order bits $h.f.timer.cy$ count the cycles of a slot. Unless the timer is synchronized, slots have locally $T$ cycles, thus the low order bits are part of a modulo-$T$ counter. The high-order bits $h.f.timer.s$ count the slot index $s$ of the current slot $(r, s)$ modulo $ns$. The timer is initialized with the value $(ns - 1, T - 1)$.

The timers on all ECUs but $ECU_{send(0)}$ stall when reaching the maximum value $(ns-1, T-1)$ and wait for synchronization. The timer on $ECU_{send(0)}$ always continues counting. Details regarding the synchronization mechanism are given in Section 12.2.

The overflow signal $ovf(h)$ between the low order and the high-order bits of the counter can essentially serve as the timer interrupt signal $ti(h)$ generated by the interface hardware:[11]

$$ti(h^i) = ovf(h^i) \wedge \neg ovf(h^{i-1})$$

The low order bit of the slot counter keeps track of the parity of the current slot and is called the hardware parity signal:

$$par(h) = h.f.timer.s[0]$$

In general the $fbus$ side of the interface will see the copies $h.f.sb(par(h))$ and $h.f.rb(par(h))$. Messages are always transmitted between these two copies of the buffers. The processor on the other hand writes to $h.f.sb(\neg par(h))$ and reads from $h.f.rb(\neg par(h))$. This does not work at boundaries of rounds unless the number of slots $ns$ is even.

The configuration registers are written immediately after reset / power-up. They contain in particular the locally relevant portions of the scheduling function. Thus if $ECU_v$ is (locally) in a slot with slot index $s$ and $send(s) = v$ then $ECU_v$ will transmit the content of the send buffer $h.f.sb(par(h))$ via the $fbus$ during some transmission interval $[ts(r, s), te(r, s)]$. A serial interface that is not actively transmitting during slot $(r, s)$ puts by construction the idle value (the bit 1) on the bus.

If we can guarantee that during the transmission interval *all* ECUs are locally in slot $(r, s)$, then transmission will be successful by Lemma 6. The clock synchronization algorithm together with an appropriate choice of the transmission interval will guarantee exactly that.

[11] The interrupt signal is kept active until it is cleared by software; the extra hardware is simple.

### 12.2. Clock Synchronization

The idea of clock synchronization is easily explained: Imagine one slot is one hour and one round is one day. Assume different clocks drift by up to $drift = 5$ minutes per day. ECUs synchronize to the first bit of the message transmission due between midnight and 1 o'clock. Assume adjusting the clocks at the receiving ECUs takes up to $adj = 1$ minute. Then the maximal deviation during 1 day is $\mathit{off} = drift + adj = 6$ minutes. $ECU_{send(s)}$, which is the sender in hour $s$, is on the safe side if it starts transmitting from $s$ o'clock plus $\mathit{off}$ minutes until $\mathit{off}$ minutes before $s+1$ o'clock, i.e. somewhen in between $s:06$ o'clock and $s+1:54$ o'clock.

At midnight life becomes slightly tricky: $ECU_{send(0)}$ waits until it can be sure that everybody believes that midnight is over and hence nobody is transmitting, i.e. until its local time $0:06$. Then it starts sending. All other ECUs are waiting for the broadcast message and adjust their clocks to midnight + $\mathit{off} = 0:06$ once they detect the first falling bit. Since that might take the receiving ECUs up to 1 minute it might be $0:07$ o'clock on the sender when it is $0:06$ o'clock at the receiver; thus after synchronization the clocks differ by at most $adj = 1$ minute.

We formalize this idea in the following way: Assume without loss of generality that $send(0) = 0$. All ECUs but $ECU_0$ synchronize to the transmission start sequence (TSS) of the first message of $ECU_0$. When ECU's waiting for synchronization $(h.f.timer = (ns-1, T-1))$ receive this TSS, they advance their local slot counter to 0 and their cycle counter to $\mathit{off}$. Analysis of the algorithm will imply that for all $v \neq 0$, $ECU_v$ will be waiting for synchronization, when $ECU_0$ starts message transmission in any slot $(r, 0)$.

First we define the start times $\alpha_v(r,s)$ of slot $(r,s)$ on $ECU_v$. This is the start time of the first cycle $t$ in round $r$ when the timer in the previous cycle had the value:

$$h^{t-1}.f.timer = ((s-1 \bmod ns), T-1)$$

This is the cycle immediately after the local timer interrupts. For every round $r$, we also define the cycles $\beta_v(r)$ when the synchronization is completed on $ECU_v$. Formally this is defined as the first cycle $\beta > \alpha_v(r,0)$ such that the local timer has value:

$$h^{\beta}.f.timer = (0, \mathit{off})$$

Timing analysis of the synchronization process in the complete hardware design shows that for all $v$ and $y$ adjustment of the local timer of $ECU_v$ to value $(0, \mathit{off})$ is completed within an adjustment time $ad = 15 \cdot \tau_y$ after $\alpha_0(r,0)$:

$$\begin{aligned}\beta_0(r) &= \alpha_0(r,0) + \mathit{off} \cdot \tau_0 \\ \beta_v(r) &\leq \beta_0(r) + 15 \cdot \tau_y\end{aligned}$$

For $s \geq 1$ no synchronization takes place and the start of new slots is only determined by the progress of the local timer:

$$\alpha_v(r,s) = \begin{cases} \beta_v(r) + (T - \mathit{off}) \cdot \tau_v & s = 1 \\ \alpha_v(r,s-1) + T \cdot \tau_v & s \geq 2 \end{cases}$$

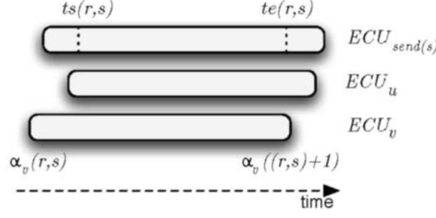



**Figure 19.** Schedules

$ECU_0$ synchronizes the other ECUs. Thus the start of slot $(r, 0)$ on $ECU_0$ depends only on the progress of the local counter:

$$\alpha_0(r, 0) = \alpha_0(r-1, ns-1) + T \cdot \tau_0$$

An easy induction on $s$ bounds the difference between start times of the same slot on different ECUs:

$$\begin{aligned} \alpha_x(r,s) - \alpha_v(r,s) &\leq 15 \cdot \tau_v + (s \cdot T - \mathit{off}) \cdot (\tau_x - \tau_v) \\ &\leq 15 \cdot \tau_v + (ns \cdot T \cdot \Delta \cdot \tau_v) \\ &= \tau_v \cdot (15 + (ns \cdot T \cdot \Delta)) \\ &= \tau_v \cdot \mathit{off} \end{aligned} \tag{3}$$

Thus we have $\mathit{off} = ad + drift$ with $ad = 15$ and $drift = ns \cdot T \cdot \Delta$.

The transmission is started in slots $(r, s)$ by $ECU_{send(s)}$ when the local cycle count is $\mathit{off}$. Thus the transmission start time is:

$$ts(r, s) = \alpha_{send(s)}(r, s) + \mathit{off} \cdot \tau_{send(s)}$$

By Lemma 6 the transmission ends at time:

$$\begin{aligned} te(r,s) &= ts(r,s) + tc \cdot \tau_{send(s)} \\ &= \alpha_{send(s)}(r,s) + (\mathit{off} + tc) \cdot \tau_{send(s)} \end{aligned}$$

The transmission interval $[ts(r, s), te(r, s)]$ must be contained in the time interval, when all ECUs are in slot $(r, s)$, as depicted in Figure 19.

**Lemma 7 (No Bus Contention)** *For all indices $v$ and $u$ of ECUs:*

$$\begin{aligned} \alpha_v(r,s) &\leq ts(r,s) \\ te(r,s) &\leq \alpha_u((r,s)+1) \end{aligned}$$

The first inequality holds because of (3). Let $x = send(s)$:

$$\begin{aligned} \alpha_v(r,s) &\leq \alpha_x(r,s) + \tau_x \cdot \mathit{off} \\ &= ts(r,s) \end{aligned}$$

The second inequality determines the minimal size of $T$:

$$\begin{aligned} te(r,s) &\leq \alpha_x(r,s) + (\mathit{off} + tc) \cdot \tau_x \\ &\leq \alpha_u(r,s) + \mathit{off} \cdot \tau_u + (\mathit{off} + tc) \cdot (1+\Delta) \cdot \tau_u \\ &\leq \alpha_u((r,s)+1) \\ &= \alpha_u(r,s) + T \cdot \tau_u \end{aligned}$$

Further calculations are necessary at the borders between rounds. Details can be found in [Pau05].

From the local start times of slots $\alpha_v(r,s)$ we calculate the numbers of local start cycles $t_v(r,s)$ using (2)

$$\alpha_v(r,s) = c_v + t_v(r,s) \cdot \tau_v$$

and then solving for $t_v(r,s)$. Trivially the number $u_v(r,s)$ of the locally last cycle on $ECU_v$ is:

$$u_v(r,s) = t_v((r,s)+1) - 1$$

Consider slot $(r,s)$. Lemma 6 and Lemma 7 then imply that the value of the send buffer of $ECU_{send(s)}$ on the network side ($par = s \bmod 2$) at the start of slot $(r,s)$ is copied to all receive buffers on the network side by the end of that slot.

**Theorem 6 (Message Transfer With Cycles)** *Let $x = send(s)$. Then for all $v$:*

$$h_x^{t_x(r,s)}.f.sb(s \bmod 2) = h_v^{u_v(r,s)}.f.rb(s \bmod 2)$$

This lemma talks only about digital hardware and hardware cycles. Thus we have shown the correctness of data transmission via the bus *and* we are back in the digital world.

## 13. Integrating f-Interfaces into the ISA

In Section 4.3 we have developed an ISA model for processors with generic devices. So far we have collected many device specific results for ECUs connected by an $fbus$. Hence there is not terribly much left to be done in order to integrate f-Interfaces into the ISA.

### 13.1. Specifying Port RAM

If a processor accesses a device $f$ with $K$ I/O byte ports, then for $k = \lceil \log K \rceil$ the device configuration (here $ecu.f$) contains a byte RAM:

$$ecu.f.m : \{0,1\}^k \to \{0,1\}^8$$

In our case the memory of the device contains the send buffer, the receive buffer –each with $\ell$ bytes where $\ell$ is a multiple of 4– and say $k$ configuration registers. Thus:

$$K = 2 \cdot \ell + 4 \cdot k$$

We use the first $\ell$ bytes of this memory for the send buffer, the next $\ell$ bytes for the receive buffer and the remaining bytes for the configuration registers. We formalize this by defining for all indices of message bytes $y \in \{0, \ldots, \ell - 1\}$:

$$sb(ecu)(y) = ecu.f.m(y)$$
$$rb(ecu)(y) = ecu.f.m(\ell + y)$$

In the absence of timer interrupts the ports are quiet. Thus, as long as no timer interrupt occurs, we can use the generic ISA model from Section 2.

*13.2. Timer Interrupt and I/O*

As pointed out earlier, at the ISA level the timer interrupt must be treated as an oracle input $eev$. Furthermore we have to deal with the external data input $fdin$ to the f-interface. If we denote by $eev^i$ the oracle input and by $fdin^i$ the external data input for the $i$-th instruction, then we get computations $ecu^0, ecu^1, \ldots$ by defining (again straight from the automata theory textbooks):

$$ecu^{i+1} = \delta_D(ecu^i, eev^i, fdin^i)$$

Within our programming model we now introduce names $j_v(r, s)$ for certain indices of local *instructions* on $ecu_v$. Intuitively, the timer interrupts the instruction executed in local configuration $ecu_v^{j_v(r,s)}$, and this locally ends slot $(r, s)$. By the results of Section 3, this is the instruction scheduled in the write back stage $WB$ in the last cycle $u_v(r, s)$, as defined in Section 12.2, of slot $(r, s)$ on $ecu_v$:

$$j_v(r, s) = s(WB, u_v(r, s)) \tag{4}$$

Note that in every cycle an instruction is scheduled in every stage. Nevertheless, due to pipeline bubbles, the write back stage might be empty in cycle $u_v(r, s)$. In this situation the scheduling functions, by construction, indicates the next instruction to arrive which is not there presently. We require interrupt event signals be only cleared by software, hence the hardware interrupt signal will stay active in cycles following $u_v(r, s)$. Thus Equation (4) also holds in this case.

This was the crucial step to get from the cycle level to the instruction level. Purely within the ISA model we continue to define:

- $i_v(r, s) = j_v((r, s) - 1) + 1$: The index of the first local instruction in slot $(r, s)$.
- $d_v(r, s) = ecu_v^{i_v(r,s)}$: The first local ISA configuration in slot $(r, s)$.
- $e_v(r, s) = ecu_v^{j_v(r,s)}$: The last local ISA configuration in slot $(r, s)$.

We can even define the sequence $eev(r, s)$ of oracle timer inputs $eev^i$ where $i \in \{i_v(r, s), \ldots, j_v(r, s)\}$. It has the form

$$eev(r, s) = 1^a 0^b 1$$

where the timer interrupt is cleared by software instruction $i_v(r, s) + a - 1$ and $a + b + 1 = j_v(r, s) - i_v(r, s) + 1$ is the number of local instructions in slot $(r, s)$.

Indeed we can complete, without any effort, the entire ISA programming model. The effect of an interrupt on the processor configuration has been defined in the previous section, thus we get for instance:

$$d_v(r,s).d.dpc = 0^{32}$$
$$d_v(r,s).d.pc = 0^{30}10$$

Also for the transition from $e_v(r,s)$ to $d_v((r,s)+1)$ and only for this transition we use the external input:

$$fdin^{j_v(r,s)} \in \{0,1\}^{8\cdot\ell}$$

Thus we assume that it consists of an entire message and we copy that message into the user-visible receive buffer

$$rb(d_v((r,s)+1)) = fdin^{j_v(r,s)}$$

Of course we also know what this message should be: The content of the user-visible send buffer of $ecu_{send(s)}$ at the end of slot $(r,s)-1$:

$$fdin^{j_v(r,s)} = sb(e_{send(s)}((r,s)-1))$$

**Theorem 7 (Buffer Broadcast)**

$$\forall v : rb(d_v((r,s)+1)) = sb(e_{send(s)}((r,s)-1))$$

This completes the user-visible ISA model. And with Theorem 6 we essentially already completed the hardware correctness proof of the implementation of Equation (1). The nondeterminism is completely encapsulated in the numbers $j_v(r,s)$ as it should be, at least if the local computations are fast enough. All we need to do is to justify the model by a hardware correctness theorem and to identify the conditions under which it can be used.

### *13.3. Hardware Correctness of the Parallel System*

For a single slot $(r,s)$, and a single processor with an f-interface, the generic hardware correctness statement from Section 4.4 translates into Theorem 8 below. Recall from Section 12.2 that we know already the start cycles $t_v(r,s)$ for all ECUs. The statement of the theorem is identical for all $ecu_v$. Thus we drop the subscript $v$. The theorem assumes that the pipe is drained and that the simulation relation between the first hardware configuration $h(r,s) = h^{t(r,s)}$ and the first ISA configuration $d(r,s) = ecu^{i(r,s)}$ of the slot holds.

**Theorem 8 (Hardware Correctness for One Slot)** *Assume that $drained(h(r,s))$ and $sim(d(r,s), h(r,s))$ holds. Then for all $t \in \{t(r,s), \ldots, (t((r,s)+1))-1\}$, for all stages $k$ and for all registers $R$ with $stage(R) = k$:*

$$\begin{aligned} h^t.p.R &= ecu^{s(k,t)}.p.R \\ m(h^t.p) &= ecu^{s(mem1,t)}.p.m \\ h^t.f.sb(\neg par(h^t)) &= sb(ecu^{s(mem1,t)}) \\ h^t.f.rb(\neg par(h^t)) &= rb(ecu^{s(mem1,t)}) \end{aligned}$$

The theorem is proven by induction over the cycles of the slot. Using the above theorem we can show:

**Theorem 9 (Hardware Correctness for System)**

$$\forall (r,s), v : drained(h_v(r,s)) \wedge sim(d_v(r,s), h_v(r,s))$$

Theorem 9 is proven by induction over the slots $(r, s)$. In order to argue about the boundaries between two slots Theorem 8 and Lemma 7 must be applied on the last cycle of the previous slot.

## 14. Pervasive Correctness Proofs

Next, we show how pervasive correctness proofs for computations with timer interrupts can be obtained from (i) correctness proofs for ISA programs that cannot be interrupted (ii) hardware correctness theorems and (iii) worst case execution time (WCET) analysis. As one would expect, the arguments are reasonably simple, but the entire formalism of the last sections is needed in order to formulate them.

We consider only programs of the form:[12]

$$\{P;\ a:\ \text{jump}\ a;\ a+4:\ \text{NOP}\}$$

The program does the useful work in portion $P$ and then waits in the idle loop for the timer interrupt. $P$ initially has to clear and then to unmask the timer interrupt, which is masked when $P$ is started (see Section 2.3).

### *14.1. Computation Theory*

We have to distinguish carefully between the transition function $\delta_D(ecu, eev, fdin)$ of the interruptible ISA computation and the transition function $\delta_U(ecu)$ of the non interruptible ISA computation that we define as follows:

$$\delta_U(ecu) = \delta_D(ecu, 0, *)$$

Observe that this definition permits the non interruptible computation to clear the timer interrupt bit by software. Non interruptible computations starting from configuration $ecu$ are obtained by iterated application of $\delta_U$:

$$\delta_U^i(ecu) = \begin{cases} ecu & i = 0 \\ \delta_U(\delta_U^{i-1}(ecu)) & \text{otherwise} \end{cases}$$

[12]Note that we have an byte addressable memory and that in an ISA with delayed branch the idle loop has two instructions.

For the ISA computation

$$d(r,s) = ecu^{i(r,s)}, ecu^{i(r,s)+1}, \dots, ecu^{j(r,s)} = e(r,s)$$

that has been constructed in Theorem 8 we get:

**Lemma 8** *For all instructions in a given slot, i.e.* $t \in [0 : (j(r,s) - i(r,s))]$*:*

$$ecu^{i(r,s)+t} = \delta_U^t(d(r,s))$$

This lemma holds due to the definition of $j(r,s)$ and the fact that the timer is masked initially such that the instructions of the interruptible computation are not interrupted.

We define the ISA run time $T_U(ecu, a)$, i.e. the time until the idle loop is reached, simply as the smallest $i$ such that $\delta_U^i$ fetches an instruction from address $a$:

$$T_U(ecu, a) = \min\{i \mid \delta_U^i(ecu).p.dpc = a\}$$

Furthermore we define the result of the non interruptible ISA computation by:

$$res_U(ecu, a) = \delta_U^{T_U(ecu,a)}(ecu)$$

Correctness proofs for non interruptible computations can be obtained by classical program correctness proofs. They usually have the form $ecu \in E \Rightarrow res_U(ecu, a) \in Q$ or, written as a Hoare triple $\{E\}P\{Q\}$.

We assume that the definition of $Q$ does not involve the PC and the delayed PC. Because the idle loop only changes the PC and the delayed PC of the ISA computation we can infer on the ISA level that property $Q$ continues to hold while we execute the idle loop:

$$\forall i \geq T_U(ecu, a) : \delta_U^i(ecu) \in Q$$

*14.2. Pervasive Correctness*

Let $sim(ecu, h)$ hold, then the ISA configuration $ecu$ can be decoded from the hardware configuration by a function:

$$ecu = decode(h)$$

Clearly, in order to apply the correctness statement $\{E\}P\{Q\}$ to a local computation in slot $(r,s)$, we have to show for the first ISA configuration in the slot:

$$d(r,s) \in E$$

Now consider the last hardware configuration $g(r,s) = h^{u(r,s)}$ of the slot $(r,s)$. We want to conclude:

**Theorem 10** *Assume the simulation relation holds initially, i.e.* $sim(d(r,s), h(r,s))$*. Then the decoded configuration obeys the postcondition Q:*

$$decode(g(r,s)) \in Q$$

This only works if portion $P$ of the program is executed fast enough on the pipelined processor hardware.

*14.3. Worst Case Execution Time*

We consider the set $H(E)$ of all hardware configurations $h$ encoding an ISA configuration $ecu \in E$:

$$H(E) = \{h \mid decode(h) \in E\}$$

While the decoding is unique, the encoding is definitely not. Portions of the ISA memory can be kept in the caches in various ways.

Given a hardware configuration $h = h^0$ we define the hardware run time $T_H(h, a)$ until a fetch from address $a$ as the smallest number of cycles such that in cycle $t$ an instruction, which has been fetched in an earlier cycle $t' < t$ from address $a$, is in the write back stage $WB$. Using scheduling functions this definition is formalized as:

$$T_H(h, a) = \min\{t' \mid \exists t : s(WB, t') = s(IF, t) \wedge h^t.dpc = a\}$$

Thus for ISA configurations satisfying $E$ we define the worst case execution time $WCET(E, a)$ as the largest hardware runtime $T_H(h, a)$ of a hardware configuration encoding a configuration in $E$:

$$WCET(E, a) = \max\{T_H(h, a) \mid h \in H(E)\}$$

As pointed out earlier such estimates can be obtained from (sound!) industrial tools based on the concept of abstract interpretation [Abs]. AbsInt's WCET analyzer does not calculate the "real" worst-case execution time $WCET(E, a)$, but an upper bound $WCET'(E, a) \geq WCET(E, a)$. Nevertheless this is sufficient for correctness since $WCET'(E, a) \leq T - \mathit{off} \Rightarrow WCET(E, a) \leq T - \mathit{off}$. Assume we have:

$$WCET(E, a) \leq T - \mathit{off}$$

Within slot $(r, s)$ we look at the ISA configuration $d(r, s) = ecu^{i(r,s)}$ and a local computation starting in hardware configuration $h(r, s) = h^{t(r,s)}$. Considering the computation after hardware run time many cycles $T_H(h(r, s), a) < T - \mathit{off}$ we can conclude that the computation is not interrupted and the instruction in the write back stage (at the end of the computation) is the first instruction being fetched from $a$. By the definition of the ISA run time this is exactly instruction $i(r, s) + T_U(d(r, s), a)$, thus we conclude:

$$s(WB, t(r, s) + T_H(h(r, s), a)) = i(r, s) + T_U(d(r, s), a)$$

Let $h' = h^{t(r,s)+T_H(h(r,s),a)}$ be the hardware configuration in this cycle and let $ecu' = ecu^{i(r,s)+T_U(d(r,s),a)} = res_U(d(r, s), a)$ be the ISA configuration of the instruction in the write back stage.

In this situation the pipe is almost drained. It contains nothing but instructions from the idle loop. Thus the processor correctness theorem $sim(ecu', h')$ holds for all components of the configuration but the PC and the delayed PC. Therefore we weaken the

simulation relation $sim$ to a relation $dsim$ by dropping the requirement that the PCs and delayed PCs should match:

$$dsim(ecu', h')$$

Until the end of the slot in cycle $t(r,s)+T$ and instruction $j(r,s)$, only instructions from the idle loops are executed. They do not affect the $dsim$ relation, hence:

$$dsim(e(r,s), g(r,s))$$

Since $res_U(d(r,s),a) \in Q$ and $Q$ does not depend on the program counters we have $e(r,s) \in Q$. We derive that $decode(g(r,s))$ coincides with $e(r,s)$ except for the program counters. And again, because this does not affect the membership in $Q$, we get the desired Theorem 10.

## 15. The Distributed OSEKtime-Like Operating System D-OLOS

### *15.1. D-OLOS Configuration*

We consider $p$ electronic control units $ECU_i$, where $i \in [0:p-1]$. On each $ECU_i$ there are $n_i$ user processes $UP(i,j)$, where $j \in [0:n_i-1]$, running under the real-time operating system OLOS. These user programs are compiled C0 programs. We denote the source program for $UP(i,j)$ by $C(i,j)$.

On each $ECU_i$ application programs $C(i,j)$ can access a set of messages buffers $MB(i)$ via system calls. Messages come in $nm$ many different types. For $k \in [0:nm-1]$ messages of type $k$ are stored in message buffers $MB(i)(k)$. Thus each $ECU_i$ is capable of storing one message of each type in its message buffers $MB(i)(k)$. These message buffers are the direct counterparts of the $FTCom$ buffers in OSEK. However we do not support fault tolerance, yet.

Messages between different ECU's are exchanged via an $fbus$ using f-interfaces. The drivers for these interfaces are part of OLOS.

As before time is divided into rounds $r$ each consisting of a fixed number $ns$ of slots $s$. From the point of view of a C0 application programmer a D-OLOS configuration $dolos$ represents the global state of the distributed system having the following components:

- $dolos.C(i,j)$ is the configuration of an abstract C0 machine representing application program $C(i,j)$ for $i \in [0:p-1]$ and $j \in [0:n_i-1]$.
- $dolos.MB(i)(k)$ is the $k$-th message in the message buffer of $ECU_i$.
- $dolos.s$ is the current slot index.
- $dolos.bus$ holds the message value of the message currently being broadcast.

### *15.2. Scheduling and Communication*

For slots $(r, s)$ we denote by $D(r, s)$ resp. $E(r, s)$ the D-OLOS configuration at the start resp. at the end of slot $(r, s)$. The message on the bus is constant during each slot $(r, s)$. It equals $D(r, s).bus$.

The scheduling of all applications $C(i, j)$ as well as the inter ECU communication procedure via the $fbus$ is identical in each round $r$ and only depends on the slot index $s$. Both are determined by three functions:

- The scheduling of all applications is defined by the global scheduling function $run$, where $run(i, s) \in [0 : n_i - 1]$. For all $i$ and $s$ this function returns the index of the application being executed in slots $(r, s)$ on $ECU_i$. Thus application $C(i, run(i, s))$ is running on $ECU_i$ during slots $(r, s)$. The state of applications that are not running does not change during a slot:

$$j \neq run(i, s) \Rightarrow E(r, s).C(i, j) = D(r, s).C(i, j)$$

- As before functions $send$ with $send(s) \in [0 : p - 1]$ gives the index of the ECU sending during slots $(r, s)$.
- The function $mtype$ with $mtype(s) \in [0 : nm - 1]$ gives the type of the message transmitted over the $fbus$ during slots $(r, s)$.

The message $bus(r, s)$ is the content of message buffer with index $mtype(s)$ of $ECU_{send(s)}$ at the end of the previous slot:

$$bus(r, s) = E((r, s) - 1).MB(send(s))(mtype(s))$$

At the start of the next slot, message $bus(r, s)$ is copied into all message buffers with index $mtype(s)$:

$$\forall i : D((r, s) + 1).MB(i)(mtype(s)) = D(r, s).bus$$

### *15.3. Local Computation*

For each $ECU_i$ and slot $(r, s)$ we have to define the effect of application $C(i, run(i, s))$ on the corresponding C0 configuration $dolos.C(i, run(i, s))$ and on the local message buffers $dolos.MB(i)(k)$. Therefore we introduce *local configurations* $lc$ being a pair with the following components: A C0 configuration $lc.c$ of a local application and a set of local message buffers $lc.MB(k)$ with $k \in [0 : nm - 1]$.

We will now define a local transition function $lc' = \delta_{LC}(lc)$. The C0 programs running under the local operating system (OLOS) can read and write $MB(k)$ using two system calls:

1. $ttsend(k, msg)$: The execution of this function results in copying the value of the C0 sub-variable with identifier $msg$ into $MB(k)$. Let $K = va(lc.c, k)$ be the current values of $k$. Then:

$$lc.c.pr = ttsend(k, msg); r \Rightarrow lc'.MB(K) = va(lc.c, msg) \wedge lc'.c.pr = r$$

2. $ttrec(k, msg)$: At invocation of this function the C0 sub-variable having the identifier $msg$ is updated with the value of $MB(k)$. Let $K = va(lc.c, k)$. Then:

$$lc.c.pr = ttrec(k, msg); r \Rightarrow va(lc'.c, msg) = lc.MB(K) \wedge lc'.c.pr = r$$

OLOS offers a third call named $ttex$. An application invoking this system call indicates that it has completed its computation for the current slot and wants to return the control back to the operating system. The execution of system call $ttex$ on the local configuration is like a $NOP$:

$$lc'.c.pr = ttex; r \Rightarrow lc'.c.pr = r$$

If the program rest does not start with one of the system calls, then an ordinary C0 instruction is executed and the message buffers stays unchanged:

$$lc'.c = \delta_C(lc.c)$$

We define run time (measured in C instructions) and result of a local computation in the usual way:

$$\begin{aligned} T_C(lc) &= \min\{t \mid \exists r : (\delta^t_{LC}(lc)).pr = ttex; r\} \\ res_{LC}(lc) &= \delta^{T_C(lc)}_{LC}(lc) \end{aligned}$$

and complete the definition of the D-OLOS semantics with the help of the result of local computations. Let $j = run(i, s)$. Then:

$$(E(r, s).C(i, j), E(r, s).MB(i)) = res_{LC}(D(r, s).C(i, j), D(r, s).MB(i))$$

Let us consider a situation where the application code is wrapped by a $while$-loop and that $ttex$ is invoked only once as the last statement of the loop body:

$$while(true)\ \{\ \text{“application code”};\ ttex\ \}$$

In this case we enforce the application code to be executed once each time the application is scheduled. Intuitively, from the applications programmers point of view, the $ttex$ system call does nothing but wait till the application is scheduled again.

## 16. D-OLOS Implementation

We implement the local version OLOS of D-OLOS by specializing the abstract kernel of CVM. The only device of CVM is an f-interface. The ISA programs of the virtual machines are obtained by compiling the local application programs. Among others the abstract kernel uses the following variables and constants (i) the constant $own$ of the kernel stores the index of the local ECU (ii) C0 implementations of the functions $run$, $send$ and $mtype$ (iii) an integer variable $s$ keeping track of the current slot (iv) an array $MB[0 : nm - 1]$ capable of storing $nm$ messages.

*16.1. Invariants*

On each $cvm(i)$, $i \in [0 : p-1]$, we will run $n_i$ virtual machines, one for each application on the $i$-th ECU. An obvious simulation relation $osim(aba)(dolos, cvm)$ is parameterized by a sequence $aba$ of allocation functions $aba(i,j)$. For each $ECU_i$ we require:

1. The kernel keeps track of the D-OLOS slot:

$$va(cvm(i).c, s) = dolos.s$$

2. The application scheduled by D-OLOS is running:

$$cvm(i).cp = run(i, dolos.s)$$

3. The user processes of CVM encode the applications of D-OLOS:

$$\forall i, j < n_i : consis(aba(i,j))(dolos.C(i,j), cvm(i).vm(j))$$

4. The content of the D-OLOS message buffers are stored in the corresponding variables of the abstract kernel:

$$\forall i, k : va(cvm(i).c, MB[k]) = dolos.MB(i)(k)$$

To argue about slot boundaries we need to define for all ECU indices $i$ and slots $(r,s)$ the first CVM configuration $dcvm(i)(r,s)$ and the last CVM configuration $ecvm(i)(r,s)$ of $cvm(i)$ in slot $(r,s)$. Slot boundaries are defined by timer interrupts.

Because the CVM primitive $wait$ is interruptible by timer interrupts, one has to extend the sequence $eev(i)^t$ of oracle interrupt event signals also for the situation, when the current process of CVM is the abstract kernel, i.e. a C0 program, and the program rest starts with $wait$. Now we have to construct a sequence $eev(i)^t$ such that the simulation theorem works. Since the kernel computation gets stuck if the program rest starts with the $wait$ primitive we can easily show that: If user processes on the $i$-th ECU are not interrupted during slot $(r,s)$ then $dcvm(i)(r,s)$ is the first configuration in slot $(r,s)$ such that $cvm(i).c.pr = wait; r'$ for some $r'$. The first configuration after the timer interrupt is defined in a similar way as the cvm configuration after a trap instruction in Section 8.4.

At slot boundaries two more 'communication' invariants are needed:

1. If $i = send(s)$, then the send buffer $sb(ecvm(i)((r,s)-1))$ on the $i$-th ECU at the end of the previous slot is the message on $dolos.bus$ during slot $(r,s)$:

$$i = send(s) \Rightarrow sb(ecvm(i)((r,s)-1)) = D(r,s).bus$$

2. The receive buffer $rb(dcvm(i)(r,s))$ on every ECU at the beginning of slot $(r,s)$ is the message on $dolos.bus$ during the previous slot:

$$\forall i : rb(dcvm(i)(r,s)) = D((r,s)-1).bus$$

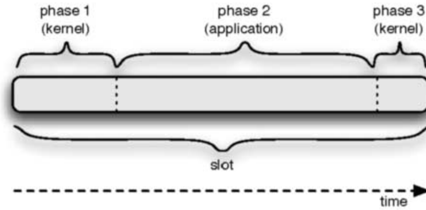



**Figure 20.** Slots in OLOS

### 16.2. Construction of the Abstract OLOS Kernel

Assume that all invariants hold for slot $(r, s) - 1$ we construct the abstract OLOS kernel such that they are maintained during slot $(r, s)$. One round of a CVM computation on an ECU proceeds in three phases as shown in Figure 20. In phases 1 and 3 the kernel runs; in phase 2 a user process runs and invokes system calls. The following happens in phase 1:

1. The kernel is running and increments $s$. Hence part 1 of $osim$ holds.
2. A driver using variants of the CVM *copy* primitive copies the local receive buffer into variable $MB[type(own, s - 1)]$. This implies that part 4 of $osim$ holds after phase 1.
3. The next process to be started is computed by $cup = run(own, s)$. Furthermore the CVM primitive $start(cup)$ is executed. Hence part 2 of $osim$ holds after phase 1.

During phase 2 we only have to worry about the running process. It is easy to implement the handlers for system calls $ttsend$ and $ttrec$ with the help of the CVM *copy* primitive such that parts 3 and 4 of $osim$ hold. Phase 2 ends by a system call $ttex$ of the application returning control to the kernel again. The kernel determines if the ECU is the sender in the next slot:

$$send(s + 1) =^{?} own$$

If this is the case it copies the content of variable $MB[mtype(s + 1)]$ into the local send buffer. This implies part 1 of the communication invariant. In any case the kernel then executes the $wait$ primitive and idles waiting for the end of the round.

Worst case execution time analysis for an ECU must consider all assembler programs running on the ECU: The compiled concrete kernel as well as the compiled user programs. Address $a$ from Section 14 is the address, where the compiled concrete kernel starts waiting for the timer interrupt. Theorem 7 then implies part 2 of the communication invariant.

## 17. The Auto Focus Task Model (AFTM)

### 17.1. Configurations

The AutoFocus task model (AFTM) is a computational model for a restricted version of the AutoFocus CASE tool [Aut]. The restrictions aim at making the implementation of

AFTM by D-OLOS efficient. As in many high level CASE tools AFTM programs are modeled by a certain number $M$ of communicating 'task' automata $T(i)$. For technical reasons we add a automaton $T(M+1)$ that models the environment and always generates output. We number the automata with indices $i \in [1 : M + 1]$.

Each $T(i)$ has $nip(i)$ input ports $IP(i)(j)$ with $j \in [0 : nip(i)-1]$ as well as $nop(i)$ output ports $OP(i)(j)$ with $j \in [0 : nop(i) - 1]$. A function $src$ (for source) specifies for each input port $IP(i)(j)$ the index $(i', j') = src(i, j)$ of the output port such that $OP(i')(j')$ is connected to $IP(i)(j)$. An AFTM configuration $aftm$ has the following components:

- $aftm.S(i)$: The state of the $i$'th task automaton. It is split into a control component $aftm.S(i).con$ and data components $aftm.S(i).x$. Each automaton has a set of control state called $idle$.
- $aftm.IP(i)(j)$: The current value of input port $IP(i)(j)$.
- $aftm.OP(i)(j)$: The current value of output port $OP(i)(j)$.

Input and output ports can hold non empty values or a special empty value $\epsilon$.

Initially (in configuration $aftm^0$) all automata are in an idle state and all ports are empty. Indices $i$ of tasks are partitioned into three classes: Indices of AND-tasks, OR tasks and the special 'environment' automaton $T(M + 1)$:

$$[1 : M + 1] = T_{and} \uplus T_{or} \uplus \{M + 1\}$$

### 17.2. Local and Global AFTM Computations

If a task $i$ is runnable in configuration $aftm$ is defined by the $runnable(aftm, i)$ predicate:

- The environment task is always runnable:

$$\forall aftm : runnable(aftm, M + 1)$$

- OR-tasks are runnable if one of their inputs is non empty:

$$i \in T_{or} \Rightarrow runnable(aftm, i) \Leftrightarrow \exists j : aftm.IP(i)(j) \neq \epsilon$$

- AND-tasks are runnable if all their inputs are non empty:

$$i \in T_{and} \Rightarrow runnable(aftm, i) \Leftrightarrow \forall j : aftm.IP(i)(j) \neq \epsilon$$

For AFTM configurations $aftm$ we define the next configuration $aftm'$ of an AFTM step. AFTM computations are then defined in the usual way by :

$$aftm^{r+1} = (aftm^r)'$$

In AFTM, each step consists of two phases. In the first phase all runnable tasks make locally a number of micro steps until an idle state is reached again. In the second phase

values of non empty output ports are copied into the connected input ports and all output ports are cleared.[13] Formalization of this model is straight forward.

The local computation is specified by a 'local AutoFocus' transition function $\delta_{LAF}$ mapping states $S$ and a vector $IP$ of input port contents to states $S'$ and a vector of output port contents $OP'$:

$$(S', OP') = \delta_{LAF}(S, IP)$$

The local run time $T_{LAF}(\mathit{aftm}, i)$ –in automata steps– of runnable task $i$ in configuration $\mathit{aftm}$ is defined by:

$$T_{LAF}(\mathit{aftm}, i) = \min\{t \mid \delta^t_{LAF}(\mathit{aftm}.S(i), \mathit{aftm}.IP(i)).S.con = \mathit{idle}\}$$

The result of this local computation is:

$$res_{LAF}(\mathit{aftm}, i) = \delta_{LAF}^{T_{LAF}(\mathit{aftm},i)}(\mathit{aftm}.S(i), \mathit{aftm}.IP(i))$$

We define the configuration $\mathit{aftm}''$ after the local computations by:

1. For runnable tasks state and output ports are determined by the result of local computations. Input ports are cleared:

$$runnable(\mathit{aftm}, i) \Rightarrow \begin{cases} (\mathit{aftm}''.S(i), \mathit{aftm}''.OP(i)) = res_{LAF}(\mathit{aftm}, i) \\ \forall j : \mathit{aftm}''.IP(i)(j) = \epsilon \end{cases}$$

2. State and output ports of non runnable tasks don't change. Input ports are not cleared. Thus new inputs will be accumulated in the communication phase:

$$\neg runnable(\mathit{aftm}, i) \Rightarrow \begin{cases} \mathit{aftm}''.T(i) = \mathit{aftm}.S(i) \\ \forall j, k : (\mathit{aftm}''.IP(i)(j), \mathit{aftm}''.OP(i)(k)) = (\mathit{aftm}.IP(i)(j), \mathit{aftm}.OP(i)(k)) \end{cases}$$

In the communication phase non empty contents of output ports are copied into connected input ports. Let $src(i, j) = (i', j')$. Then:

$$\mathit{aftm}'.IP(i)(j) = \begin{cases} \mathit{aftm}''.OP(i')(j') & \mathit{aftm}''.OP(i')(j') \neq \epsilon \\ \mathit{aftm}''.IP(i)(j) & \text{otherwise} \end{cases}$$

All output ports are cleared:

$$\forall i, j : \mathit{aftm}'.OP(i)(j) = \epsilon$$

The local state does not change during the communication phase:

$$\forall i : \mathit{aftm}'.S(i) = \mathit{aftm}''.S(i)$$

[13] An easy exercise shows that this model is equivalent to the model described in [BBG+06].

## 18. Simulation of AFTM by D-OLOS

### *18.1. C0 Code Generation for Local Computation*

A local configuration $T$ of a task automaton is a triple with the following components: State $T.S$, content of input ports $T.IP(k)$, where $k \in [0 : nip-1]$ and content of output ports $T.OP(k)$, where $k \in [0 : nip-1]$.

In order to implement a single task automaton $T$ as a process in OLOS, we first need a C0 program $prog(T)$ that simulates local runs of the automaton in the following sense:

- I/O: Inputs are read from a C array $IP[0 : nip-1]$ and outputs are written to another C-array $OP[0 : nop-1]$. Access to these arrays is restricted to assignments of the form $e = IP[e']$ for input and $OP[e'] = e$ for output operations; where $e$ and $e'$ are expressions. This restriction makes it later easy to replace these assignments by operating system calls like $ttrec(e, e')$; the replacement will however be slightly more involved.
- Data: Each data component $S.x$ of the state has its counter part in a C variables with the name $x$.

Recall that for C0 configurations $c$ and expressions $e$ we denote by $va(c, e)$ the value of expression $e$ in configuration $c$. A trivial simulation relation $asim(T, c)$ between $T$ and $c$ is established by requiring for all $j$ and $x$

$$T.OP(j) = va(c, OP[j])$$
$$T.IP(j) = va(c, IP[j])$$
$$T.x = va(c, x)$$

Assume $asim(T, c)$ holds and assume that both the automaton and the C machine are in their initial states. For the C machine this means that the program rest is the body of the main function. This body is formally to be found in the function table $c.ft$ at argument $main$.

$$T.con = idle$$
$$c.pr = c.ft(main).body$$

The program $prog(T)$ is specified by requiring that the simulation relation holds for the results of the computations:

$$asim(res_{LAF}(T), res_C(c))$$

There are several ways to produce the program $prog(T)$ from the task automaton $T$. The program could for instance be generated by hand or by a translation tool. Also the correctness proof can either be done by hand or by an automatic translation validation tool.

For a program generated by a verified generation tool, no further correctness proof would be needed. However to the best of our knowledge no such tool exists yet. Note that in any case we are dealing with plain C code verification only.

### 18.2. Deployment

We will simulate each step of AFTM by one round consisting of $ns$ slots of D-OLOS. Thus, we will be interested to relate $aftm^r$ with $D(r, 0)$. In order to deploy an AFTM machine on D-OLOS machine we have to specify several things:

- Task deployment: For each automata we have to specify the C0 application $C(i, j)$ that simulates the task. Let $p$ be the number of ECUs and $N$ the maximum number of task executable on an ECU. Then this will be done with an injective task deployment function

$$depl : [1 : M + 1] \to [0 : p - 1] \times [0 : N - 1]$$

- Application scheduling: For every ECU index $i$ and for every slot $s$ we have to specify the C0 application $run(i, s)$ running on $ECU_i$ during slot $s$:

$$run : [0 : p - 1] \times [0 : ns - 1] \to [0 : N - 1]$$

  This defines for each task $T(k)$ and round $r$ a slot $start(k) < ns$, such that task $T(k)$ is simulated in slot $start(k)$ of the round:

$$depl(k) = (i, j) \Rightarrow start(k) = s \Leftrightarrow run(i, s) = j$$

### 18.3. Output Port Broadcasting

Recall that for each AFTM task $T(i)$ we denote by $nip(i)$ resp. $nop(i)$ the number output ports resp. input ports of task $T(i)$. The set of all indices of output ports is denoted by

$$OP = \bigcup_i \{i\} \times [0 : nop(i) - 1]$$

We denote by $N$ the cardinality of this set. For each pair of indices $(i, j) \in OP$ we specify a function:

$$broad : OP \to [0 : ns - 1]$$

During each round $r$ we plan to broadcast $(aftm^r)''.OP(i)(j)$ (i.e. the content of port $OP(i)(j)$ after the local computation phase of macro step $r$ in slot $broad(i, j)$.

We require in each round, that any output port $OP(i)(j)$ of task $i$ be broadcast after the task has run:

$$\forall i, j : broad(i, j) > start(i)$$

Obviously we need $ns \geq N+1$. This is the only restriction we impose on schedules. Schedules will tend to be shorter if tasks with many output ports are scheduled earlier than tasks with few output ports.

The content of output port $OP(i)(j)$ will be stored in $MB(u)(broad(i, j))$, for all $ECU_u$. Equivalently the output port broadcast in slot $s$ is stored on the $i$-th ECU in message buffers $MB(i)(s)$.

### *18.4. Invariants*

At the slot boundaries we maintain four invariants between the AFTM configurations $aftm^r$, $(aftm^r)''$ and the corresponding D-OLOS configuration $D(r,s)$. For all indices $e$ of ECUs, for all indices $i$ and $j$ of output ports $OP(i)(j)$ and for all slots $(r,s)$:

1. Consider an output port $OP(i)(j)$ and the message buffers $MB(e)(broad(i,j))$. Before or while $OP(i)(j)$ is scheduled for the broadcast, the message buffers contain the value of $OP(i)(j)$ before the local computation phase, i.e. the value that was broadcast in the last round. Afterwards they have the value as $OP(i)(j)$ after the local computation phase. There is however an exception. On the $ECU_e$ where task $i$ is deployed (formally: $e$ is the first component of $depl(i)$) the new values are already in the local message buffers after the task has been simulated:

$$D(r,s).MB(e)(broad(i,j)) = \begin{cases} (aftm^r)''.OP(i)(j) & s > broad(i,j) \vee s > start(i) \wedge e = fst(depl(i)) \\ (aftm^{r-1})''.OP(i)(j) & \text{otherwise} \end{cases}$$

2. Consider a data component $aftm.S(i).x$ of task $i$ and the C0 variable $x$ of the application $C(depl(i))$ that simulates task $i$. Until the task is scheduled for simulation, the value of the variable is the value of $x$. Otherwise it is the value after the communication phase, which is the same as the value after the computation phase:

$$va(D(r,s).C(depl(i)),x) = \begin{cases} aftm^r(i).S.x & start(i) \le s \\ aftm^{r+1}(i).S.x & \text{otherwise} \end{cases}$$

3. The invariants given so far do not suffice to infer the input buffers $aftm.IP(i)(j)$ from the message buffers for slots $s = start(i)$. Let $(i',j') = src(i,j)$ and assume that $broad(i',j') < start(i)$. Then the output port value $(aftm^{r-1})''.OP(i')(j')$ needed for the computation of input port value $aftm^r.IP(i)(j)$ is already overwritten in the message buffers. Therefore we save in the previous round the endangered value $(aftm^{r-1})''.OP(i')(j')$ into a 'shadow message buffer' $SMB[j]$ of the application $i$. We define a predicate $q(i,j)$ stating that a shadow message buffer is needed:

$$q(i,j) \Leftrightarrow broad(src(i,j)) < start(i)$$

We require:

$$q(i,j) \Rightarrow va(D(r,s).C(depl(i)),SMB[j]) = \begin{cases} (aftm^{r-1})''.OP(i')(j') & s \le start(i) \\ (aftm^r)''.OP(i')(j') & \text{otherwise} \end{cases}$$

Initially the shadow buffers must be set to $\epsilon$.

4. Finally we must track the accumulation of values in the input ports $IP(i)(j)$. This is done in array elements $IP[j]$ of application $C(depl(i))$. Even if task $i$ is not runnable in step $r$ the array $IP$ must be updated. We require:

$$va(D(r,s).C(depl(i)), IP[j]) = \begin{cases} (aftm^{r-1}).IP(i)(j) & s \leq start(i) \\ (aftm^{r}).IP(i)(j) & \text{otherwise} \end{cases}$$

*18.5. Construction of D-OLOS Applications*

In the following we argue inductively on the current slot number. Given that application $C(depl(i))$ is starting in slot $(r, start(i))$, we first simulate the communication phase at the end of the previous slot.

The input port values $aftm^r.IP(i)(j)$ at the start of step $r$ are accumulated in C0 array $IP(i)$. Let $k(i,j) = broad(src(i,j))$ be the index of message buffers, where values of the output port $OP(src(i,j))$ connected to $IP(i)(j)$ are stored. Then for each $j$ the new value of $IP[j]$ is computed as follows. The current content of message buffer $MB(fst(depl(i))(k(i,j)))$ is accessed with a $ttrec$ system call. If $q(i,j) = 0$ non-$\epsilon$ values are stored in $IP[j]$:

$$ttrec(k(i,j), X);\ if\ (X \neq \epsilon)\ then\ IP[j] = X$$

Otherwise, if the shadow buffer $SMB[j]$ is different form $\epsilon$ it is copied into $IP[j]$. Furthermore $SMB[j]$ itself is updated with the value of $MB(fst(depl(i))(k(i,j)))$, using the $ttrec$ system-call:

$$if\ (SMB[j] \neq \epsilon)\ then\ IP[j] = SMB[j];\ ttrec(k(i,j), SMB[j])$$

In the C0 configurations $C(depl(i))$ after execution of these pieces of code we conclude from the invariants of the previous slot

$$va(C(depl(i)), IP[j]) = aftm^r.IP(i)(j)$$

and that invariant 3 holds. Then we clear all entries $OP[j]$ in the C0 array of output values. For configurations $C(depl(i))$ after the execution of this code the following holds:

$$va(C(depl(i)), OP[j]) = aftm^r.OP(i)(j) = \epsilon$$

*Local Computations.* Task $i$ tests if it is runnable. If so the program $prog(T(i))$ is run. For configurations $C(depl(i))$ after execution of this piece of code we conclude that invariant 2 holds and that array $OP$ holds the values of the output ports $OP(i)(j)$ after the local computation phase:

$$va(C(depl(i)), OP[j]) = (aftm^r)''.OP(i)(j)$$

For runnable tasks we clear the input array $IP$, for non runnable tasks, the input array stays unchanged. From this we conclude invariant 4.

*Updating the Message Buffer.* Using the $ttsend$ system call the new values of the output ports are copied into their message buffers

$$ttsend(broad(i,j), OP[j])$$

After this invariant 1 holds and we are done.

## References


[Abs] AbsInt Angewandte Informatik GmbH. `http://www.absint.com/`.

[Aut] AutoFocus Project. `http://autofocus.in.tum.de`.

[BBG+05] S. Beyer, P. Böhm, M. Gerke, M. Hillebrand, T. In der Rieden, S. Knapp, D. Leinenbach, and W.J. Paul. Towards the Formal Verification of Lower System Layers in Automotive Systems. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2–5 October 2005, San Jose, CA, USA, Proceedings*, pages 317–324. IEEE, 2005.

[BBG+06] J. Botaschanjan, M. Broy, A. Gruler, A. Harhurin, S. Knapp, L. Kof, W. Paul, and M. Spichkova. On the Correctness of Upper Layers of Automotive Systems. 2006. To appear.

[Bey05] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, March 2005.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In *Proc. of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS, pages 51–65. Springer, 2003.

[BP06] Geoffrey M. Brown and Lee Pike. Easy Parameterized Verification of Biphase Mark and 8N1 Protocols. In *Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2006.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Computer Science Department, July 2006.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the Verification of Memory Management Mechanisms. In *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.

[Fle] FlexRay Consortium. `http://www.flexray.com`.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the Correctness of Operating System Kernels. In J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Computer Science Department, June 2005.

[HIP05] Mark Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *ICCD '05*, pages 309–316. IEEE Computer Society, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HW73] C. A. R. Hoare and Niklaus Wirth. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica (ACTA)*, 2:335–355, 1973.

[Kna05] Steffen Knapp. Towards the Verification of Functional and Timely Behavior of an eCall Implementation. Master's thesis, Universität des Saarlandes, 2005.

[KP06] Steffen Knapp and Wolfgang Paul. Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, 2006. To appear.

[Lei06] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Department, 2006. To appear.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness. In Bernhard Aichernig and Bernhard Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, pages 2–11, 2005.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992, revised online version: 1999.

[Nor98] Michael Norrish. C Formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, December 1998.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OSE01a] OSEK group. *OSEK/VDX Fault-Tolerant Communication (FTCom)*, 2001. `http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf`.

[OSE01b] OSEK group. *OSEK/VDX Time-Triggered Operating System (OLOS)*, 2001. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`.

[Pau05] Wolfgang Paul. Lecture Notes from the lecture Computer Architecture 2: Automotive Systems. `http://www-wjp.cs.uni-sb.de/lehre/vorlesung/ rechnerarchitektur2/ws0506/temp/060302_CA2_AUTO.pdf`, 2005.

[Pet06] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, 2006. To appear.

[Rus94] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 304–313, New York, NY, USA, 1994. ACM Press.

[Sch87] Fred B. Schneider. Understanding Protocols for Byzantine Clock Synchronization. Technical report, Department of Computer Science, Ithaca, NY, USA, 1987.

[Sch06a] Julien Schmaltz. A Formal Model of Lower System Layer. In Aarti Gupta and Panagiotis Manolios, editors, *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, CA, USA, November 12–16, 2006, Proceedings*. IEEE Computer Society, 2006. To appear.

[Sch06b] Julien Schmaltz. A Formalization of Clock Domain Crossing and Semi-Automatic Verification of Low Level Clock Synchronization Hardware. 2006. To appear.

[SH98] Jun Sawada and Warren A. Hunt. Processor Verification with Precise Exceptions and Speculative Execution. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV '98*, pages 135–146. Springer, 1998.

[SK06] Wilfried Steiner and Hermann Kopetz. The Startup Problem in Fault-Tolerant Time-Triggered Communication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 35–44, Washington, DC, USA, 2006. IEEE Computer Society.

[VERa] The VERIFIX Project. `http://www.info.uni-karlsruhe.de/~verifix/`.

[Verb] The Verisoft Project. `http://www.verisoft.de/`.

[Win93] G. Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*

# Verification and Synthesis of Reactive Programs[1]

Amir PNUELI [2]
*New York University and Weizmann Institute of Science* [3]

**Abstract.** In these notes we present a viable approach to the synthesis of reactive programs from a temporal specification of their desired behavior. When successful, this direct correct-by-construction approach to system development obviates the need for post-facto verification.

In spite of the established double exponential lower bound that applies to the general case, we show that many useful specifications fall into a class of temporal formulas, technically identified as the Reactivity(1) class, for which we present an $n^3$ synthesis algorithm.

**Keywords.** Program synthesis, formal verification, correct-by-construction development, temporal logic, reactivity

## 1. Introduction

One of the most ambitious and challenging problems in reactive systems construction is the automatic synthesis of programs and (digital) designs from logical specifications. First identified as Church's problem [5], several methods have been proposed for its solution ([4], [17]). The two prevalent approaches to solving the synthesis problem were by reducing it to the emptiness problem of tree automata, and viewing it as the solution of a two-person game. In these preliminary studies of the problem, the logical specification that the synthesized system should satisfy was given as an S1S formula.

This problem has been considered again in [16] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL). This followed two previous attempts ([6], [13]) to synthesize programs from temporal specification which reduced the synthesis problem to satisfiability, ignoring the fact that the environment should be treated as an adversary. The method proposed in [16] for a given LTL specification $\varphi$ starts by constructing a Büchi automaton $\mathcal{B}_\varphi$, which is then determinized into a deterministic Rabin automaton. This double translation may reach complexity of double exponent in the size of $\varphi$. Once the Rabin automaton is obtained, the game can be solved in time $n^{O(k)}$, where $n$ is the number of states of the automaton and $k$ is the number of accepting pairs.

[1] Research supported in part by the European community project Prosyd, the John von-Neumann Minerva center for Verification of Reactive Systems, ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

[2] Joint work with Nir Piterma and Yaniv Sa'ar

[3] E-mail: amir@cs.nyu.edu

The high complexity established in [16] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to use it for any sizeable system development. Yet there exist several interesting cases where, if the specification of the design to be synthesized is restricted to simpler automata or partial fragments of LTL, it has been shown that the synthesis problem can be solved in polynomial time. Representative cases are the work in [3] which presents (besides the generalization to real time) efficient polynomial solutions ($N^2$) to games (and hence synthesis problems) where the acceptance condition is one of the LTL formulas $\Box p$, $\Diamond q$, $\Box \Diamond p$, or $\Diamond \Box q$. A more recent paper is [2] which presents efficient synthesis approaches for the LTL fragment consisting of a boolean combinations of formulas of the form $\Box p$.

This paper can be viewed as a generalization of the results of [3] and [2] into the wider class of *generalized Reactivity*(1) formulas (GR(1)), i.e. formulas of the form

$$(\Box \Diamond p_1 \wedge \cdots \wedge \Box \Diamond p_m) \quad \rightarrow \quad (\Box \Diamond q_1 \wedge \cdots \wedge \Box \Diamond q_n) \tag{1}$$

Following the developments in [8], we show how any synthesis problem whose specification is a GR(1) formula can be solved in time $N^3$, where $N$ is the size of the state space of the design. Furthermore, we present a (symbolic) algorithm for extracting a design (program) which implements the specification. We make an argument that the class of GR(1) formulas is sufficiently expressive to provide complete specifications of many designs.

This work has been developed as part of the Prosyd project (see www.prosyd.org) which aims at the development of a methodology and a tool suit for the property-based construction of digital circuits from their temporal specification. Within the prosyd project, synthesis techniques are applied to check first whether a set of properties is *realizable*, and then to automatically produce digital designs of smaller units.

Most of the technical material contained in this paper is taken from [14] which is a joint work with Nir Piterman and Yaniv Sa'ar.

One of the main points we wish to emphasize in this more detailed account of the techniques is the observations that the game-theoretic approach to designs synthesis can be viewed as a generalization of the iterative techniques regularly employed for model checking. The generalization can be expressed as replacing the operator for computing the *predecessor* of an assertion by the more general operator of *controlled predecessor*.

## 2. Fair Discrete Systems and their Computations

As our computational model, we take *fair discrete systems* (FDS) [10]. This generalizes the model of *fair transition systems* [12] by allowing a more general form of fairness requirements. . An FDS is represented by a tuple $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

- $V$ – A finite set of typed state variables. A $V$-state $s$ is an interpretation of $V$. Denote by $\Sigma_V$ – the set of all $V$-states.
- $\Theta$ – An initial condition. A satisfiable assertion that characterizes the initial states.
- $\rho$ – A transition relation. An assertion $\rho(V, V')$, referring to both unprimed (current) and primed (next) versions of the state variables. For example, $x' = x + 1$ corresponds to the assignment $x := x + 1$.

- $\mathcal{J} = \{J_1, \ldots, J_k\}$ A set of justice (weak fairness) requirements. Ensure that a computation has infinitely many $J_i$-states for each $J_i,\ i = 1, \ldots, k$.
- $\mathcal{C} = \{\langle p_1, q_1\rangle, \ldots \langle p_n, q_n\rangle\}$ A set of compassion (strong fairness) requirements. The compassion requirement $\langle p_i, q_i\rangle$ implies that a computation that contains Infinitely many $p_i$-states must also contain infinitely many $q_i$-states.

*A Simple Programming Language:* SPL

To illustrate reactive programs we use a simple programming language SPL. This language allows composition of parallel processes communicating by shared variables.

As an example of an SPL program we present in Fig. 1 a program consisting of two parallel processes.

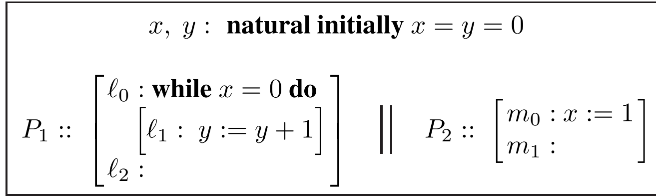



**Figure 1.** Program ANY-Y

The FDS corresponding to program ANY-Y is given by:

- State Variables $V: \begin{pmatrix} x, y : \text{natural} \\ \pi_1 \ \ : \{\ell_0, \ell_1, \ell_2\} \\ \pi_2 \ \ : \{m_0, m_1\} \end{pmatrix}.$
- Initial condition:

$$\Theta :\ \pi_1 = \ell_0 \ \wedge\ \pi_2 = m_0 \ \wedge\ x = y = 0.$$

- Transition Relation: $\rho :\ \rho_I \vee \rho_{\ell_0} \vee \rho_{\ell_1} \vee \rho_{m_0}$, with appropriate disjunct for each statement. For example, the disjuncts $\rho_I$ and $\rho_{\ell_0}$ are

$$\rho_I :\ \pi_1' = \pi_1 \ \wedge\ \pi_2' = \pi_2 \ \wedge\ x' = x \ \wedge\ y' = y$$

$$\rho_{\ell_0} : \pi_1 = \ell_0 \ \wedge\ \begin{pmatrix} x = 0 \ \wedge\ \pi_1' = \ell_1 \\ \vee \\ x \neq 0 \ \wedge\ \pi_1' = \ell_2 \end{pmatrix} \ \wedge\ \pi_2' = \pi_2 \ \wedge\ x' = x \ \wedge\ y' = y$$

- Justice set: $\mathcal{J}:\ \{\neg at_\ell_0, \neg at_\ell_1, \neg at_m_0\}$.
- Compassion set: $\mathcal{C}:\ \emptyset$.

As seen in this example, we standardly include in the set of system variables the program counters $\pi_1$ and $\pi_2$ which point to the locations, in each process, of the next statement to be executed in this process.

The transition $\rho_I$ is the *idling transition* which preserves the values of all variables. This transition is standardly included in the transition relation of any program in order to guarantee that some transition is enabled on every state, even when the program terminates.

*Computations*

Let $\mathcal{D}$ be an FDS for which the above components have been identified. The state $s'$ is defined to be a $\mathcal{D}$-successor of state $s$ if

$$\langle s, s' \rangle \models \rho_{\mathcal{D}}(V, V').$$

That is, $\rho_{\mathcal{D}}$ evaluates to *true* when we interpret every $x \in V$ as $s[x]$ and every $x'$ as $s'[x]$.

We define a computation of $\mathcal{D}$ to be an infinite sequence of states

$$\sigma : s_0, s_1, s_2, ...,$$

satisfying the following requirements:

- Initiality: $s_0$ is initial, i.e., $s_0 \models \Theta$.
- Consecution: For each $j \geq 0$, the state $s_{j+1}$ is a $\mathcal{D}$-successor of the state $s_j$.
- Justice: For each $J \in \mathcal{J}$, $\sigma$ contains infinitely many $J$-positions. This guarantees that every *just transition* is disabled infinitely many times.
- Compassion: For each $\langle p, q \rangle \in \mathcal{C}$, if $\sigma$ contains infinitely many $p$-positions, it must also contain infinitely many $q$-positions. This guarantees that every compassionate transition which is enabled infinitely many times is also taken infinitely many times.

We denote by $\mathcal{C}omp(\mathcal{D})$ the set of computations of FDS $\mathcal{D}$.

*Examples of Computations*

Identification of the FDS $\mathcal{D}_P$ corresponding to a program $P$ gives rise to a set of computations $\mathcal{C}omp(P) = \mathcal{C}omp(\mathcal{D}_P)$.

The following computation of program ANY-Y corresponds to the case that $m_0$ is the first executed statement:

$$\begin{array}{l} \langle \pi_1 : \ell_0 \, , \, \pi_2 : m_0 \; ; \; x : 0 \, , \, y : 0 \rangle \xrightarrow{m_0} \langle \pi_1 : \ell_0 \, , \, \pi_2 : m_1 \; ; \; x : 1 \, , \, y : 0 \rangle \xrightarrow{\ell_0} \\ \langle \pi_1 : \ell_2 \, , \, \pi_2 : m_1 \; ; \; x : 1 \, , \, y : 0 \rangle \xrightarrow{\tau_I} \cdots \xrightarrow{\tau_I} \cdots \end{array}$$

The following computation corresponds to the case that statement $\ell_1$ is executed before $m_0$.

$$\begin{array}{l} \langle \pi_1 : \ell_0 \, , \, \pi_2 : m_0 \; ; \; x : 0 \, , \, y : 0 \rangle \xrightarrow{\ell_0} \langle \pi_1 : \ell_1 \, , \, \pi_2 : m_0 \; ; \; x : 0 \, , \, y : 0 \rangle \xrightarrow{\ell_1} \\ \langle \pi_1 : \ell_0 \, , \, \pi_2 : m_0 \; ; \; x : 0 \, , \, y : 1 \rangle \xrightarrow{m_0} \langle \pi_1 : \ell_0 \, , \, \pi_2 : m_1 \; ; \; x : 1 \, , \, y : 1 \rangle \xrightarrow{\ell_0} \\ \langle \pi_1 : \ell_2 \, , \, \pi_2 : m_1 \; ; \; x : 1 \, , \, y : 1 \rangle \xrightarrow{\tau_I} \cdots \xrightarrow{\tau_I} \cdots \end{array}$$

In a similar way, we can construct for each $n \geq 0$ a computation that executes the body of statement $\ell_0$ $n$ times and then terminates in the final state
$\langle \pi_1 : \ell_2 \, , \, \pi_2 : m_1 \; ; \; x : 1 \, , \, y : n \rangle$.

*A Non-Computation*

While we can delay termination of the program for an arbitrary long time, we cannot postpone it forever. Thus, the sequence

$$\langle \pi_1 : \ell_0 \,,\, \pi_2 : m_0 \,;\, x : 0 \,,\, y : 0 \rangle \xrightarrow{\ell_0} \langle \pi_1 : \ell_1 \,,\, \pi_2 : m_0 \,;\, x : 0 \,,\, y : 0 \rangle \xrightarrow{\ell_1}$$
$$\langle \pi_1 : \ell_0 \,,\, \pi_2 : m_0 \,;\, x : 0 \,,\, y : 1 \rangle \xrightarrow{\ell_0} \langle \pi_1 : \ell_1 \,,\, \pi_2 : m_0 \,;\, x : 0 \,,\, y : 1 \rangle \xrightarrow{\ell_1}$$
$$\langle \pi_1 : \ell_0 \,,\, \pi_2 : m_0 \,;\, x : 0 \,,\, y : 2 \rangle \xrightarrow{\ell_0} \langle \pi_1 : \ell_1 \,,\, \pi_2 : m_0 \,;\, x : 0 \,,\, y : 2 \rangle \xrightarrow{\ell_1}$$
$$\langle \pi_1 : \ell_0 \,,\, \pi_2 : m_0 \,;\, x : 0 \,,\, y : 3 \rangle \xrightarrow{\ell_0} \cdots$$

in which statement $m_0$ is never executed is not an admissible computation. This is because it violates the justice requirement $\neg at_m_0$ contributed by statement $m_0$, by having no states in which this requirement holds.

This illustrates how the requirement of justice ensures that program ANY-Y always terminates. Justice guarantees that every (enabled) process eventually progresses, in spite of the representation of concurrency by *interleaving*.

*FDS Operations: Asynchronous Parallel Composition*

The asynchronous parallel composition of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \parallel \mathcal{D}_2$, is given by $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{lcl} V & = & V_1 \cup V_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 \\ \rho & = & (\rho_1 \wedge pres(V_2 - V_1)) \ \vee \ (\rho_2 \wedge pres(V_1 - V_2)) \\ \mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 \\ \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}$$

The predicate $pres(U)$ stands for the assertion $U' = U$, implying that all the variables in $U$ are preserved by the transition.

As implied by the definition, a step taken by $\mathcal{D}_1 \parallel \mathcal{D}_2$ is a step that is taken by either system $\mathcal{D}_1$ or system $\mathcal{D}_2$, while preserving the variables local to the other system.

Asynchronous parallel composition represents the interleaving-based concurrency which is assumed in shared-variables models.

**Claim 1** $\mathcal{D}(P_1 \parallel P_2) \quad \sim \quad \mathcal{D}(P_1) \parallel \mathcal{D}(P_2)$

Thus, given an SPL program $P_1 \parallel P_2$, we can either compute the FDS corresponding to the entire program, or compute separately $\mathcal{D}(P_1)$ and $\mathcal{D}(P_2)$ and then take their asynchronous parallel composition. Claim 1 assures us that the resulting FDS will be the same in both cases.

*Synchronous Parallel Composition*

The synchronous parallel composition of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \parallel\!| \mathcal{D}_2$, is given by the FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{lcl} V & = & V_1 \cup V_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 \\ \rho & = & \rho_1 \wedge \rho_2 \\ \mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 \\ \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}$$

As implied by the definition, a step taken by $\mathcal{D}_1 \mathbin{|||} \mathcal{D}_2$ consists of jointly taking a $\mathcal{D}_1$-step together with a $\mathcal{D}_2$-step.

Synchronous parallel composition can be used for hardware verification, where it is the natural operator for combining two circuits into a composed circuit. Here we use it for model checking of LTL formulas.

**Claim 2** *A sequence $\sigma$ of $V$-states is a computation of the combined $\mathcal{D}_1 \mathbin{|||} \mathcal{D}_2$ iff $\sigma \Downarrow_{V_1}$ is a computation of $\mathcal{D}_1$ and $\sigma \Downarrow_{V_2}$ is a computation of $\mathcal{D}_2$.*

Here, $\sigma \Downarrow_{V_i}$ denotes the sequence obtained from $\sigma$ by restricting each of the states to a $V_i$-state, i.e. projecting the states on the variables $V_i$.

*Feasibility and Viability of Systems*

An FDS $\mathcal{D}$ is said to be *feasible* if $\mathcal{D}$ has at least one computation.

An infinite sequence of states is defined to be a *run* of an FDS $\mathcal{D}$ if it satisfies the requirements of initiality and consecution but not necessarily any of the fairness requirements.

The FDS $\mathcal{D}$ is defined to be *viable* if any finite run of $\mathcal{D}$ can be extended to a computation of $\mathcal{D}$.

**Claim 3** *Every* FDS *derived from an* SPL *program is viable.*

Note that if $\mathcal{D}$ is a viable system, such that its initial condition $\Theta_{\mathcal{D}}$ is satisfiable, then $\mathcal{D}$ is feasible.

## 3. The Specification Language

As the language for specifying properties of reactive systems we use *linear-time temporal logic* (LTL) [11].

*Requirement Specification Language: Temporal Logic*

Assume an underlying (first-order) assertion language. The predicate $at_\ell_i$, abbreviates the formula $\pi_j = \ell_i$, where $\ell_i$ is a location within process $P_j$.

A temporal formula is constructed out of state formulas (assertions) to which we apply the boolean operators $\neg$ and $\vee$ and the basic temporal operators:

| | | | |
|---|---|---|---|
| $\bigcirc$ | – Next | $\ominus$ | – Previous |
| $\mathcal{U}$ | – Until | $\mathcal{S}$ | – Since |

Other temporal operators can be defined in terms of the basic ones as follows:

| | |
|---|---|
| $\diamondsuit\, p = \mathrm{T}\,\mathcal{U} p$ | – Eventually |
| $\square\, p = \neg \diamondsuit \neg p$ | – Henceforth |
| $p\,\mathcal{W}\,q = \square\, p \vee (p \mathcal{U} q)$ | – Waiting-for, Unless, Weak Until |
| $\diamondminus\, p = \mathrm{T}\,\mathcal{S} p$ | – Sometimes in the past |
| $\boxminus\, p = \neg \diamondminus \neg p$ | – Always in the past |
| $p\,\mathcal{B}\,q = \boxminus\, p \vee (p \mathcal{S} q)$ | – Back-to, Weak Since |

A model for a temporal formula $p$ is an infinite sequence of states $\sigma : s_0, s_1, ...,$ where each state $s_j$ provides an interpretation for the variables of $p$.

*Semantics of* LTL

Given a model $\sigma$, we define the notion of a temporal formula $p$ holding at a position $j \geq 0$ in $\sigma$, denoted by $(\sigma, j) \models p$:

- For an assertion $p$,
  $(\sigma, j) \models p \iff s_j \models p$
  That is, we evaluate $p$ locally on state $s_j$.
- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \iff (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ for some $k \geq j, (\sigma, k) \models q$,
  and for every $i$ such that $j \leq i < k, \quad (\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \iff j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p \mathcal{S} q \iff$ for some $k \leq j, (\sigma, k) \models q$,
  and for every $i$ such that $j \geq i > k, \quad (\sigma, i) \models p$

This implies the following semantics for the derived operators:

- $(\sigma, j) \models \Box p \iff (\sigma, k) \models p$ for all $k \geq j$
- $(\sigma, j) \models \Diamond p \iff (\sigma, k) \models p$ for some $k \geq j$

If $(\sigma, 0) \models p$ we say that $p$ holds over $\sigma$ and write $\sigma \models p$. Formula $p$ is *satisfiable* if it holds over some model. Formula $p$ is (temporally) *valid* if it holds over all models.

Formulas $p$ and $q$ are *equivalent*, denoted $p \sim q$, if $p \leftrightarrow q$ is valid. They are called *congruent*, denoted $p \approx q$, if $\Box(p \leftrightarrow q)$ is valid. If $p \approx q$ then $p$ can be replaced by $q$ in any context.

The *entailment* $p \Rightarrow q$ is an abbreviation for $\Box(p \rightarrow q)$.

For an FDS $\mathcal{D}$ and an LTL formula $\varphi$, we say that $\varphi$ is $\mathcal{D}$-*valid*, denoted $\mathcal{D} \models \varphi$, if all computations of $\mathcal{D}$ satisfy $\varphi$.

*Reading Exercises*

Following are some temporal formulas $\varphi$ and a verbal formulation of the constraint they impose on a state sequence $\sigma : s_0, s_1, \ldots$ such that $\sigma \models \varphi$:

- $p \rightarrow \Diamond q$ — If $p$ holds at $s_0$, then $q$ holds at $s_j$ for some $j \geq 0$.
- $\Box(p \rightarrow \Diamond q)$ — Every $p$ is followed by a $q$. Can also be written as $p \Rightarrow \Diamond q$.
- $\Box \Diamond q$ — The sequence $\sigma$ contains infinitely many $q$'s.
- $\Diamond \Box q$ — All but finitely many states in $\sigma$ satisfy $q$. Property $q$ eventually stabilizes.
- $q \Rightarrow \diamondminus p$ — Every $q$ is preceded by a $p$ — *causality*.
- $(\neg r)\ \mathcal{W}\ q$ — $q$ *precedes* $r$. $r$ cannot occur before $q$ — precedence. Note that $q$ is not guaranteed, but $r$ cannot happen without a preceding $q$.
- $(\neg r)\ \mathcal{W}\ (q \wedge \neg r)$ — $q$ *strongly precedes* $r$.
- $p \Rightarrow (\neg r)\ \mathcal{W}\ q$ — Following every $p$, $q$ precedes $r$.

*Classification of Formulas/Properties*

A formula of the form $\Box p$ for some past formula $p$ is called a *safety* formula.

A formula of the form $\Box \Diamond p$ for some past formula $p$ is called a response formula. An equivalent characterization is the form $p \Rightarrow \Diamond q$. The equivalence is justified by

$$\Box(p \rightarrow \Diamond q) \qquad \sim \qquad \Box \Diamond((\neg p)\ \mathcal{B}\ q)$$

Both formulas state that either there are infinitely many $q$'s, or there there are no $p$'s, or there is a last $q$-position, beyond which there are no further $p$'s.

A property is classified as a *safety/response* property if it can be specified by a *safety/response* formula.

Every temporal formula is equivalent to a conjunction of a reactivity formulas, i.e.

$$\bigwedge_{i=1}^{k} (\Box \Diamond p_i \ \vee \ \Diamond \Box q_i)$$

*Hierarchy of the Temporal Properties*

In Fig. 2 we present a hierarchy of the temporal properties. Every box in this diagram represents a class of properties together with the canonical formula corresponding to this class. The formulas $p$, $p_i$, $q$, $q_i$ appearing in the canonical representations are arbitrary *past formulas*. Lines connecting the boxes in the diagram represent strict inclusion relations between the classes. Thus, the class of *safety properties* is strictly included in the class of *obligation properties*. This means that every safety property is also an obligation property, but there exists an obligation property which is not a safety property. Note that, the obligation and reactivity classes contain each an internal strict hierarchy parameterized by $k$.

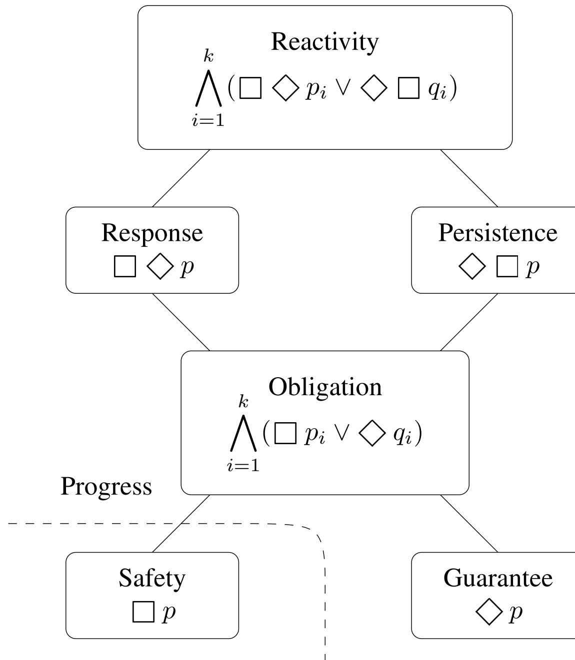



**Figure 2.** The Temporal Hierarchy of Properties

*Temporal Specification of Properties*

Formula $\varphi$ is $\mathcal{D}$-valid, denoted $\mathcal{D} \models \varphi$, if all *initial states* of $\mathcal{D}$ satisfy $\varphi$. Such a formula specifies a property of $\mathcal{D}$.

We illustrate these notions on program MUX-SEM presented in Fig. 3. This program implements mutual exclusion by the use of semaphores.

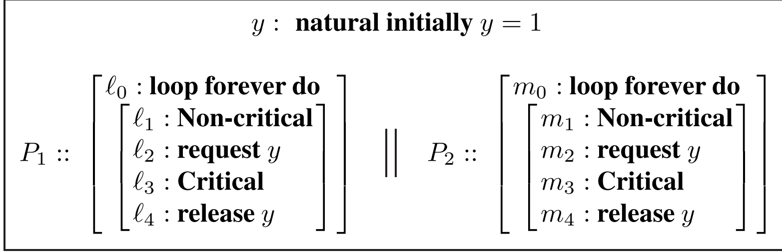


**Figure 3.** Program MUX-SEM

The main properties we wish to specify for this program are:

- Mutual Exclusion — No computation of the program can include a state in which both processes are in their respective critical sections, i.e., process $P_1$ is at $\ell_3$ while $P_2$ is at $m_3$. This property can be specified by the formula

$$\square \neg(at_\ell_3 \ \wedge \ at_m_3)$$

- Accessibility for $P_1$ — Whenever process $P_1$ is at $\ell_2$, it shall eventually reach its critical section at $\ell_3$. This property is specifiable by the formula

$$\square(at_\ell_2 \quad \rightarrow \quad \diamond\, at_\ell_3)$$

- Accessibility for $P_2$ — A similar requirement for Process $P_2$. This property is specifiable by the formula

$$\square(at_m_2 \quad \rightarrow \quad \diamond\, at_m_3)$$

## 4. Model Checking

In this section we will present an approach to the algorithmic verification of finite-state reactive systems. We refer the reader to [7] for a more comprehensive discussion of model checking and the various approaches to its implementations.

*Model Checking*

Model checking is a process by which we algorithmically check that a given finite state FDS $D$ satisfies its temporal specification $\varphi$. There are two approaches to this process:

- Enumerative (explicit state) approach, by which we construct a graph containing all the reachable states of the system, and then apply *graph theoretic* algorithms to its analysis.
- Symbolic approach, by which we continuously work with assertions which characterize sets of states.

Here, we consider the *symbolic* approach. Note that every assertion over a finite-domain FDS can be represented as a boolean formula over boolean variables. Assume that a finite-state FDS is represented by such formulas, including the *initial condition* $\Theta$ and the bi-assertion $\rho$ representing the *transition relation*.

We assume that we have an efficient representation of boolean assertions, and efficient algorithms for manipulation of such assertions, including all the boolean operations as well as existential and universal quantification. Note that, for a boolean variable $b$,

$$\exists b : \varphi(b) = \varphi(0) \ \vee \ \varphi(1) \qquad\qquad \forall b : \varphi(b) = \varphi(0) \ \wedge \ \varphi(1)$$

Also assume that we can efficiently check whether a given assertion is *valid*, i.e., equivalent to $1$ (*True*).

*The Essence of Model Checking*

All that one needs to know in order to perform general model checking, is how to check for reachability and response. That is, verifying the properties $\Box\, p$ and $p \Rightarrow \Diamond\, q$.

Once these capabilities are attained, one can model check all of LTL.

*Predecessors and Their Transitive Closure*

For an assertion $\varphi(V)$ and a bi-assertion $R(V, V')$, we define the existential predecessor predicate transformer:

$$R \diamond \psi \quad = \quad \exists V' : R(V, V') \wedge \psi(V')$$

Obviously

$$\|R \diamond \varphi\| \quad = \quad \{s \mid s \textit{ is an } R\textit{-predecessor of a } \varphi\textit{-state}\}$$

For example

$$(x' = x + 1) \diamond (x = 1) \quad = \quad \exists x' : x' = x + 1 \ \wedge \ x' = 1 \quad \sim \quad x = 0$$

The immediate predecessor transformer can be iterated to yield the eventual predecessor transformer:

$$R^* \diamond \varphi \qquad = \quad \varphi \ \vee \ R \diamond \varphi \ \vee \ R \diamond (R \diamond \varphi) \ \vee \ R \diamond (R \diamond (R \diamond \varphi)) \ \vee \ \cdots$$

*Formulation as Fixed Points*

Consider a recursive equation of the general form $y = f(y)$, where $y$ is an assertion representing a set of states. Such an equation is called a *fix-point* equation.

Not every fix-point equation has a solution. For example, the equation $y = \neg y$ has no solution.
The assertional expression $f(y)$ is called *monotonic* if it satisfies the requirement

$$\|y_1\| \subseteq \|y_2\| \quad \textit{implies} \quad \|f(y_1)\| \subseteq \|f(y_2)\|$$

*Solutions to Fix-point Equations*

Every assertional expression $f(y)$ which is constructed out of the assertion variable $y$ and arbitrary constant assertions, to which we apply the boolean operators $\vee$ and $\wedge$, and the *predecessor operator* $\rho \diamond p$ is monotonic.
Consider a fix-point equation

$$y = f(y) \tag{2}$$

It may have $0$, one, or many solutions. For example, the equation $y = y$ has many solutions. A solution $y_m$ is called a *minimal solution* if it satisfies $\|y_m\| \subseteq \|y\|$ for any solution $y$ of Equation (2). A solution $y_{\scriptscriptstyle M}$ is called a *maximal solution* if it satisfies $\|y_{\scriptscriptstyle M}\| \supseteq \|y\|$ for any solution $y$ of Equation (2). We denote by $\mu y.f(y)$ and $\nu y.f(y)$ the minimal and maximal solutions, respectively.

**Claim 4** *If $f(y)$ is a monotonic expression, then the fix-point equation $y = f(y)$ has both a minimal and a maximal solution which can be obtained by the iteration sequence*

$$y_1 = f(y_0),\ y_2 = f(y_1),\ y_3 = f(y_2),\ \ldots$$

*where $y_0 = 0$ for the minimal solution, and $y_0 = 1$ for the maximal solution.*

*Expressing the Eventual Predecessor*

The eventual predecessor can be expressed by a minimal fix-point expression:

$$\rho^* \diamond q \quad = \quad \mu y.(q \ \vee \ \rho \diamond y)$$

This is because the fix-point expression generates the following approximation sequence:

$$\begin{array}{l}
y_0 = 0 \\
y_1 = q \ \vee \ 0 \qquad\ \ = q \\
y_2 = q \ \vee \ \rho \diamond y_1 = q \ \vee \ \rho \diamond q \\
y_3 = q \ \vee \ \rho \diamond y_2 = q \ \vee \ \rho \diamond q \ \vee \ \rho \diamond (\rho \diamond q) \\
\qquad\qquad\qquad\quad \cdots
\end{array}$$

Characterizing the set of all states which initiate a path leading to a $q$-state.

*A Symbolic Algorithm for Model Checking Invariance*

**Algorithm** INV $(\mathcal{D}, p)$ : **assertion** — Check that FDS $\mathcal{D}$ satisfies *Inv*$(p)$, using *symbolic operations*

$new$ : **assertion**

1. $new := 0$
2. **Fix** $(new)$ **do**
3. $\quad new := \neg p \vee (\rho_{\mathcal{D}} \diamond new)$
4. **return** $\Theta_{\mathcal{D}} \wedge new$

where

$$\textbf{Fix } (y) \textbf{ do } S \quad = \quad old := \neg y\textbf{; While } (y \neq old) \textbf{ do [} old := y\textbf{; } S\textbf{]}$$

The algorithm returns an assertion characterizing all the initial states from which there exists a finite path leading to violation of $p$. It returns the empty (*false*) assertion iff $\mathcal{D}$ satisfies *Inv*$(p)$.
An equivalent formulation is

$$\textbf{return } \Theta_{\mathcal{D}} \wedge \mu y. \neg p \vee \rho_{\mathcal{D}} \diamond y$$

*Checking for Feasibility*

Before we discuss model checking response properties we consider the problem of checking whether a given FDS is feasible.

Recall that a *run* of an FDS is an infinite sequence of states which satisfies the requirements of initiality and consecution but not necessarily any of the fairness requirements.

A state $s$ of an FDS $\mathcal{D}$ is called *reachable* if it participates in some run of $\mathcal{D}$.
A state $s$ is called *feasible* if it participates in some computation. The FDS is called *feasible* if it has at least one computation.

A set of states $S$ is defined to be an *F-set* if it satisfies the following requirements:

- F1. All states in $S$ are reachable.
- F2. Each state $s \in S$ has a $\rho$-successor in $S$.
- F3. For every state $s \in S$ and every justice requirement $J \in \mathcal{J}$, there exists a path leading from $s$ to some $J$-state in $S$.
- F4. For every state $s \in S$ and every compassion requirement $(p, q) \in \mathcal{C}$, either there exists an $S$-path leading from $s$ to some $q$-state, or $s$ satisfies $\neg p$.

**Claim 5 (F-sets)**
*A reachable state $s$ is feasible* iff *it has a path leading to some F-set.*

*Proof:*
Assume that $s$ is a feasible state. Then it participates in some computation $\sigma$. Let $S$ be the (finite) set of all states that appear infinitely many times in $\sigma$. We will show that $S$ is an F-set. It is not difficult to see that there exists a cutoff position $t \geq 0$ such that $S$ contains all the states that appear at positions beyond $t$.

Obviously all states appearing in $\sigma$ are reachable. If $s \in S$ appears in $\sigma$ at position $i > t$ then it has a successor $s_{i+1} \in \sigma$ which is also a member of $S$.

Let $s = s_i \in \sigma$, $i > t$ be a member of $S$ and $J \in \mathcal{J}$ be some justice requirement. Since $\sigma$ is a computation it contains infinitely many $J$-positions. Let $k \geq i$ one of the $J$-positions appearing later than $i$. Then the path $s_i, \ldots, s_k$ is an $S$-path leading from $s$ to a $J$-state.

Let $s = s_i \in \sigma$, $i > t$ be a member of $S$ and $(p, q) \in \mathcal{C}$ be some compassion requirement. There are two possibilities by which $\sigma$ may satisfy $(p, q)$. Either $\sigma$ contains only finitely many $p$-positions, or $\sigma$ contains infinitely many $q$ positions. It follows that either $S$ contains no $p$-states, or it contains some $q$-states which appear infinitely many times in $\sigma$. In the first case, $s$ satisfies $\neg p$. In the second case, there exists a path leading from $s_i$ to $s_k$, a $q$-state such that $k \geq i$.

In the other direction, assume the existence of an F-set $S$ and a reachable state $s$ which has a path leading to some state $s_1 \in S$. We will show that there exists a computation $\sigma$ which contains $s$.

Since $s$ is reachable and has a path leading to state $s_1 \in S$, there exists a finite sequence of states $\pi$ leading from an initial state to $s_1$ and passing through $s$. We will show how $\pi$ can be extended to a computation by an infinite repetition of the following steps. At any point in the construction, we denote by $\mathit{end}(\pi)$ the state which currently appears last in $\pi$.

- We know that $\mathit{end}(\pi) \in S$ has a successor $s \in S$. Append $s$ to the end of $\pi$.
- Consider in turn each of the justice requirements $J \in \mathcal{J}$. We append to $\pi$ the $S$-path $\pi_J$ connecting $\mathit{end}(\pi)$ to a $J$-state.
- Consider in turn each of the compassion requirements $(p, q) \in \mathcal{C}$. If there exists an $S$-path $\pi_q$, connecting $\mathit{end}(\pi)$ to a $q$-state, we append $\pi_q$ to the end of $\pi$. Otherwise, we do not modify $\pi$. We observe that if there does not exist an $S$-path leading from $\mathit{end}(\pi)$ to a $q$-state, then $\mathit{end}(\pi)$ and all of its progeny within $S$ must satisfy $\neg p$.

It is not difficult to see that the infinite sequence constructed in this way is a computation. ∎

*Computing F-Sets*

Assume an assertion $\varphi$ which characterizes an F-set. Translating requirements F1–F4 into formulas, we obtain the following implications:

| | | | |
|---|---|---|---|
| $\varphi$ | $\rightarrow$ | $\mathit{reachable}_{\mathcal{D}}$ | |
| $\varphi$ | $\rightarrow$ | $\rho \diamond \varphi$ | Every $\varphi$-state has a $\varphi$-successor |
| $\varphi$ | $\rightarrow$ | $\mu Y.(J \wedge \varphi \;\vee\; \rho \diamond Y)$ | For every $J \in \mathcal{J}$, every $\varphi$-state has a path leading to a $J \wedge \varphi$-state |
| $\varphi$ | $\rightarrow$ | $\neg p \;\vee\; (\varphi \wedge \rho)^* \diamond (\varphi \wedge q)$ | For every $(p, q) \in \mathcal{C}$ and each $\varphi$-state $s$, either $s \models \neg p$, or $s$ initiates a path leading to a $q \wedge \varphi$-state |

This can be summarized as

$$\varphi \quad \rightarrow \quad \left( \begin{array}{l} \mathit{reachable}_{\mathcal{D}} \quad \wedge \;\; \rho \diamond \varphi \quad \wedge \\ \bigwedge\limits_{J \in \mathcal{J}} (\varphi \wedge \rho)^* \diamond (\varphi \wedge J) \;\wedge\; \bigwedge\limits_{(p,q) \in \mathcal{C}} \neg p \;\vee\; (\varphi \wedge \rho)^* \diamond (\varphi \wedge q) \end{array} \right)$$

Since we are interested in a maximal F-set, the computation can be expressed as:

$$\nu\varphi.\left(\begin{array}{l} \textit{reachable}_{\mathcal{D}} \qquad\qquad\qquad\quad \wedge\ \rho \diamond \varphi \qquad \wedge \\ \bigwedge\limits_{J\in\mathcal{J}} (\varphi\wedge\rho)^* \diamond(\varphi\wedge J)\ \wedge \bigwedge\limits_{(p,q)\in\mathcal{C}} \neg p\ \vee\ (\varphi\wedge\rho)^*\diamond(\varphi\wedge q) \end{array}\right)$$

*Algorithmic Interpretation*

Computing the maximal fix-point as a sequence of iterations, we can describe the computational process as follows:
Start by letting $\varphi := \textit{reachable}_{\mathcal{D}}$. Then repeat the following steps:

- Remove from $\varphi$ all states which do not have a $\varphi$-successor.
- For each $J \in \mathcal{J}$, remove from $\varphi$ all states which do not have a path leading to a $(J \wedge \varphi)$-state.
- For each $(p, q) \in \mathcal{C}$, remove from $\varphi$ all $p$-states which do not have a $\varphi$-path leading to a $q$-state.

until no further change.

To check whether an FDS $\mathcal{D}$ is feasible, we compute for it the maximal F-set and check whether it is empty. $\mathcal{D}$ is feasible iff the maximal F-set is not-empty.

*Example*

As an example, consider the FDS presented in Fig. 4.

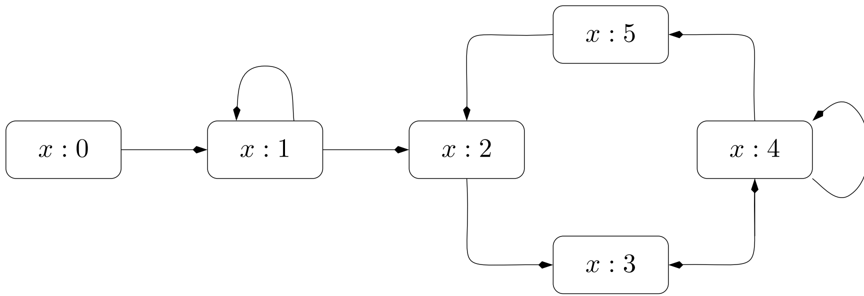



**Figure 4.** An Example FDS

This system is associated with the following fairness requirements:

$$\begin{array}{l} J_1 : x \neq 1 \\ C_1 : (x = 3,\ x = 5) \\ C_2 : (x = 2,\ x = 1) \end{array}$$

We set $\varphi_0 : \{0..5\}$ and then proceed as follows:

- Removing from $\varphi_0$ all $(x = 2)$-states that do not have a $\varphi_0$-path leading to an $(x = 1)$-state, as required by $C_2$, we are left with $\varphi_1 : \{0, 1, 3, 4, 5\}$.
- Removing from $\varphi_1$ all states that do not have a path leading to a $(x \neq 1)$-state, as required by $J_1$, leaves $\varphi_2 : \{0, 3, 4, 5\}$.

- Successively removing from $\varphi_2$ all states without successors, leaves $\varphi_3 : \{3, 4\}$.
- Removing from $\varphi_3$ all $(x = 3)$-states which do not have a $\varphi_2$-path leading to a $(x = 5)$-state, as required by $C_1$, we are left with $\varphi_4 : \{4\}$.
- No reasons to remove any further states from $\varphi_4 : \{4\}$, so this is our final set.

We conclude that the FDS of Fig. 4 is *feasible*, under the assumption that all states are initial.

*Verifying Response Properties Through Feasibility Checking*

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be an FDS and $p \Rightarrow \diamondsuit\, q$ be a response property we wish to verify over $\mathcal{D}$. Let $\mathit{reachable}_{\mathcal{D}}$ be the assertion characterizing all the reachable states in $\mathcal{D}$.
We define an auxiliary FDS $\mathcal{D}_{p,q} : \langle V, \Theta_{p,q}, \rho_{p,q}, \mathcal{J}, \mathcal{C}\rangle$, where

$$\Theta_{p,q} : \mathit{reachable}_{\mathcal{D}} \;\wedge\; p \;\wedge\; \neg q$$
$$\rho_{p,q} :\; \rho \;\wedge\; \neg q'$$

Thus, $\Theta_{p,q}$ characterizes all the $\mathcal{D}$-reachable $p$-states which do not satisfy $q$, while $\rho_{p,q}$ allows any $\rho$-step as long as the successor does not satisfy $q$.

**Claim 6 (Model Checking Response)**
$\mathcal{D} \models p \Rightarrow \diamondsuit\, q$ iff $\mathcal{D}_{p,q}$ *is infeasible.*

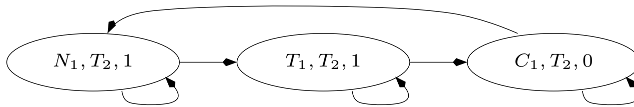
*Proof:* The claim is justified by the observation that every computation of $\mathcal{D}_{p,q}$ can be extended to a computation of $\mathcal{D}$ that violates the response property $p \Rightarrow \diamondsuit\, q$. Indeed, let $\sigma : s_k, s_{k+1}, \ldots$ be a computation of $\mathcal{D}_{p,q}$. By the definition of $\Theta_{p,q}$, we know that $s_k$ is a $\mathcal{D}$-reachable $p$-state. Thus, there exists, a finite sequence $s_0, \ldots, s_k$, such that $s_0$ is $\mathcal{D}$-initial. The infinite sequence $s_0, \ldots, s_{k-1}, s_k, s_{k+1}, \ldots$ is a computation of $\mathcal{D}$ which contains a $p$-state at position $k$, and has no following $q$-state. This sequence violates $p \Rightarrow \diamondsuit\, q$. ▟

*Example: a Simpler* MUX-SEM

In Fig. 5, we present a simpler version of program MUX-SEM.
The semaphore instructions **request** $y$ and **release** $y$ respectively stand for

$$\langle \textbf{when}\ y = 1\ \textbf{do}\ y := 0\rangle \quad \text{and} \quad y := 1.$$

In Fig. 6 we present the set of all reachable states of program SIMPLE-MUXSEM. Assume we wish to verify the property $T_2 \Rightarrow \diamondsuit\, C_2$. We start by forming the FDS SIMPLE-MUXSEM $_{T_2,C_2}$, whose set of reachable states is given by:




First, we eliminate all $(T_2 \wedge y = 1)$-states which do not have a path leading to a $C_2$-state. This leaves us with an FDS consisting of a single state, as presented in Fig. 7.

Next, we eliminate all states which do not have a path leading to a $\neg C_1$-state. This leaves us with nothing. We conclude that MUX-SEM $\models T_2 \Rightarrow \diamondsuit\, C_2$.

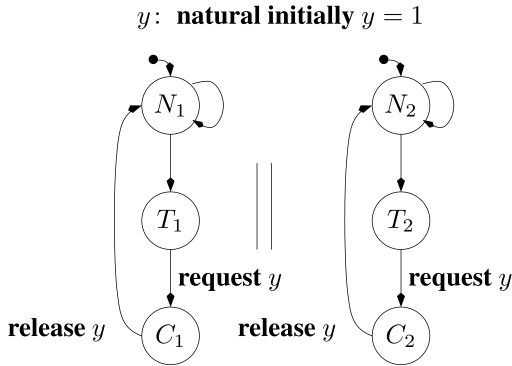



**Figure 5.** Program SIMPLE-MUXSEM— a 3-Location Version of MUX-SEM

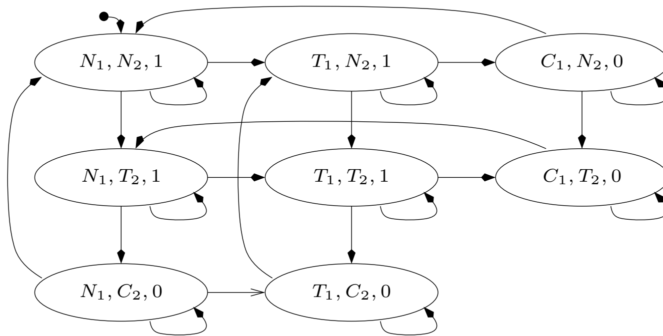



**Figure 6.** Set of reachable states of Program SIMPLE-MUXSEM

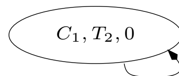



**Figure 7.** A single-state FDS

## 5. Temporal Testers

In this section we present the construction of temporal testers and explain how they are used for model checking of a general LTL formula. For a more extensive discussion of temporal testers we refer the reader to [9].

For every LTL formula $\varphi$, there exists an FDS $T[\varphi]$ called the *temporal tester* for $\varphi$. This tester has a distinguished boolean variable $x$ such that, in every $\sigma$ a computation of $T[\varphi]$, and every position $j \geq 0$, $x[s_j] = 1$ iff $(\sigma, j) \models \varphi$. In such a case, we say that $x$ *matches* $\varphi$ *in* $\sigma$.

We can view $T[\varphi]$ as a (possibly non-deterministic) *transducer* which incrementally reads the values of the variables occurring in the formula $\varphi$ and outputs in $x$ the current value of $\varphi$ over the infinite sequence.

### *5.1. Construction of Temporal Testers*

A formula $\varphi$ is called a *principally temporal formula* (PTF) if the main operator of $p$ is temporal. With no serious loss of generality, we restrict our attention to LTL formulas which are PTF's.

A PTF is called a *basic temporal formula* if it contains no other PTF as a proper subformula. We start our construction by presenting temporal testers for the basic temporal formulas.

*A Tester for $\bigcirc p$*

The tester for the formula $\bigcirc p$ is given by:

$$T[\bigcirc p] : \begin{cases} V : \mathit{Vars}(p) \cup \{x\} \\ \Theta : 1 \\ \rho : x = p' \\ \mathcal{J} = \mathcal{C} : \emptyset \end{cases}$$

**Claim 7**
*$T[\bigcirc p]$ is a temporal tester for $\bigcirc p$.*

*Proof:*
Let $\sigma$ be a computation of $T[\bigcirc p]$. We will show that $x$ matches $\bigcirc p$ in $\sigma$. Let $j \geq 0$ be any position. By the transition relation, $x = 1$ at position $j$ iff $s_{j+1} \models p$ iff $(\sigma, j) \models \bigcirc p$.

Let $\sigma$ be an infinite sequence such that $x$ matches $\bigcirc p$ in $\sigma$. We will show that $\sigma$ is a computation of $T[\bigcirc p]$. For any position $j \geq 0$, $x = 1$ at $j$ iff $(\sigma, j) \models \bigcirc p$, iff $s_{j+1} \models p$. Thus, $x$ satisfies $x = p'$ at every position $j$. ▟

*A Tester for $p\mathcal{U}q$*

The tester for the formula $p\mathcal{U}q$ is given by:

$$T[p\mathcal{U}q] : \begin{cases} V : \mathit{Vars}(p, q) \cup \{x\} \\ \Theta : 1 \\ \rho : x = q \vee (p \wedge x') \\ \mathcal{J} : q \vee \neg x \\ \mathcal{C} : \emptyset \end{cases}$$

**Claim 8** *$T[p\mathcal{U}q]$ is a temporal tester for $p\mathcal{U}q$.*

*Proof:*
Let $\sigma$ be a computation of $T[p\mathcal{U}q]$. We will show that $x$ matches $p\mathcal{U}q$ in $\sigma$. Let $j \geq 0$ be any position. Consider first the case that $s_j \models x$ and we will show that $(\sigma, j) \models p\mathcal{U}q$. According to the transition relation, $s_j \models x$ implies that either $s_j \models q$ or $s_j \models p$ and $s_{j+1} \models x$. If $s_j \models q$ then $(\sigma, j) \models p\mathcal{U}q$ and we are done. Otherwise, we apply the same argument to position $j + 1$. Continuing in this manner, we either locate a $k \geq j$ such that $s_k \models q$ and $s_i \models p$ for all $i, j \leq i < k$, or we have $s_i \models \neg q \wedge p \wedge x$ for all $i \geq j$. If we locate a stopping $k$ then, obviously $(\sigma, j) \models p\mathcal{U}q$ according to the semantic definition of the $\mathcal{U}$ operator. The other case in which both $\neg q$ and $x$ hold over all positions beyond $j$ is

impossible since it violates the justice requirement demanding that $\sigma$ contains infinitely many positions at which either $q$ is true or $x$ is false.

Next we consider the case that $\sigma$ is a computation of $T[p\mathcal{U}q]$ and $(\sigma, j) \models p\mathcal{U}q$, and we have to show that $s_j \models x$. According to the semantic definition, there exists a $k \geq j$ such that $s_k \models q$ and $s_i \models p$ for all $i, j \leq i < k$. Proceeding from $k$ backwards all the way down to $j$, we can show (by induction if necessary) that the transition relation implies that $s_t \models x$ for all $t = k, k-1, \ldots, j$.

In the other direction, let $\sigma$ be an infinite sequence such that $x$ matches $p\mathcal{U}q$ in $\sigma$. We will show that $\sigma$ is a computation of $T[p\mathcal{U}q]$. From the semantic definition of $\mathcal{U}$ it follows that $(\sigma, j) \models p\mathcal{U}q$ iff either $s_j \models q$ or $s_j \models p$ and $(\sigma, j+1) \models p\mathcal{U}q$. Thus, if $x = (p\mathcal{U}q)$ at all positions, the transition relation $x = q \;\vee\; (p \wedge x')$ holds at all positions. To show that $x$ satisfies the justice requirement $q \vee \neg x$ it is enough to consider the case that $\sigma$ contains only finitely many $q$-positions. In that case, there must exist a cutoff position $c \geq 0$ such that no position beyond $c$ satisfies $q$. In this case, $p\mathcal{U}q$ must be false at all positions beyond $c$. Consequently, $x$ is false at all positions beyond $c$ and is therefore false at infinitely many positions. ▟

*Why Do We Need the Justice Requirement?*

Reconsider the definition of the temporal tester for $p\mathcal{U}q$. We will show that the justice requirement $q \vee \neg x$ is essential for the correctness of the construction. Consider a state sequence $\sigma : s_0, s_1, \ldots$ in which $q$ is identically false and $p$ is identically true at all positions. Obviously, $(\sigma, j) \not\models p\mathcal{U}q$, for all $j \geq 0$, and the transition relation reduces to the equation

$$x = x'.$$

This equation has two possible solutions, one in which $x$ is identically false and the other in which $x$ is identically true at all positions. Only $x = 0$ matches $p\mathcal{U}q$. This is also the only solution which satisfies the justice requirement.

Thus, the role of the justice requirement is to select among several solutions to the transition relation equation, a unique one which matches the basic temporal formula at all positions.

*A Tester for $p\,\mathcal{W}\,q$*

A supporting evidence for the significance of the justice requirements is provided by the tester for the formula $p\,\mathcal{W}\,q$:

$$T[p\,\mathcal{W}\,q] : \begin{cases} V : \mathit{Vars}(p,q) \;\cup\; \{x\} \\ \Theta : 1 \\ \rho : x \quad = \quad q \;\vee\; (p \;\wedge\; x') \\ \mathcal{J} : \neg p \;\vee\; x \\ \mathcal{C} : \emptyset \end{cases}$$

Note that the transition relation of $T[p\,\mathcal{W}\,q]$ is identical to that of $T[p\mathcal{U}q]$, and they only differ in their respective justice requirements.

The role of the justice requirement in $T[p\,\mathcal{W}\,q]$ is to eliminate the solution $x = 0$ over a computation in which $p = 1$ and $q = 0$ at all positions.

*Testers for the Derived Operators*

Based on the testers for $\mathcal{U}$ and $\mathcal{W}$, we can construct testers for the derived operators $\diamondsuit$ and $\square$. They are given by

$$T[\diamondsuit\, p] : \begin{cases} V : \mathit{Vars}(p) \;\cup\; \{x\} \\ \Theta : 1 \\ \rho : x \quad = \quad p \;\vee\; x' \\ \mathcal{J} : p \;\vee\; \neg x \\ \mathcal{C} : \emptyset \end{cases} \qquad T[\square\, p] : \begin{cases} V : \mathit{Vars}(p) \;\cup\; \{x\} \\ \Theta : 1 \\ \rho : x \quad = \quad p \;\wedge\; x' \\ \mathcal{J} : \neg p \;\vee\; x \\ \mathcal{C} : \emptyset \end{cases}$$

A formula such as $\diamondsuit\, p$ can be viewed as a "promise for an eventual $p$". The justice requirement $p \vee \neg x$ can be interpreted as suggesting:

*Either fulfill all your promises or stop promising.*

Note that once $x = 0$ in the tester $T[\diamondsuit\, p]$, it remains 0 and requires $p = 0$ ever after.

*Testers for the Basic Past Formulas*

The following are testers for the basic past formulas $\ominus\, p$ and $p\,\mathcal{S}\,q$:

$$T[\ominus\, p] : \begin{cases} V : \mathit{Vars}(p) \;\cup\; \{x\} \\ \Theta : x = 0 \\ \rho : x' \quad = \quad p \\ \mathcal{J} : \emptyset \\ \mathcal{C} : \emptyset \end{cases} \qquad T[p\,\mathcal{S}\,q] : \begin{cases} V : \mathit{Vars}(p, q) \;\cup\; \{x\} \\ \Theta : x = q \\ \rho : x' \quad = \quad q' \;\vee\; (p' \wedge x) \\ \mathcal{J} : \emptyset \\ \mathcal{C} : \emptyset \end{cases}$$

Note that testers for past formulas are not associated with any fairness requirements. On the other hand, they have a non-trivial initial condition.

### *5.2. Testers for Compound Temporal Formulas*

Up to now we only considered testers for basic formulas. The construction for non-basic formulas is based on the following reduction principle. Let $f(\varphi)$ be a temporal formula containing one or more occurrences of the basic formula $\varphi$. Then the temporal tester for $f(\varphi)$ can be constructed according to the following recipe:

$$T[f(\varphi)] \quad = \quad T[f(x_\varphi)] \;|||\; T[\varphi]$$

where, $x_\varphi$ is the boolean output variable of $T[\varphi]$, and $f(x_\varphi)$ is obtained from $f(\varphi)$ by replacing every instance of $\varphi$ by $x_\varphi$.
Following this recipe the temporal tester for an arbitrary formula $f$ can be decomposed into a synchronous parallel composition of smaller testers, one for each basic formula nested within $f$.

*Testers as Circuits*

Having viewed testers as *transducers*, we can view their composition as a *circuit interconnection*. For example, in Fig. 8 we show how a tester for the compound formula $\varphi\,\mathcal{U}\,\psi$ can be constructed by interconnecting the testers for $\varphi$, $\psi$, and the tester for the basic formula $p\,\mathcal{U}\,q$.

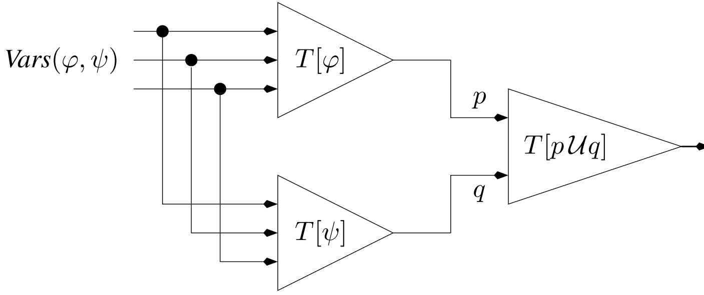



**Figure 8.** A tester for a compound formula presented as a Circuit

### 5.3. Model Checking General Temporal Formulas

To check whether $\mathcal{D} \models \varphi$, perform the following steps:

- *Construct* the tester $T[\varphi]$.
- *Form* the combined system $C = \mathcal{D} \;|||\; T[\varphi] \;|||\; \langle \Theta : \neg x_\varphi, 1, \emptyset, \emptyset \rangle$, where $\langle \Theta : \neg x_\varphi, 1, \emptyset, \emptyset \rangle$ is the trivial FDS which imposes the constraint that all initial states falsify $x_\varphi$.
- *Check* whether $C$ is feasible.
- *Conclude* $\mathcal{D} \models \varphi$ iff $C$ is infeasible.

*Example*

Consider system $\mathcal{D}$ presented in Fig. 9.

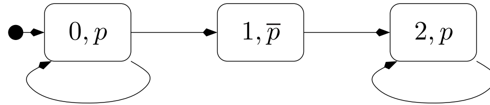



**Figure 9.** Example system $\mathcal{D}$

For which we wish to verify the property $\diamondsuit \square p$.

Composing the system with $T[\diamondsuit \square p] \;|||\; \langle \Theta : \overline{x}_\diamondsuit \cdots \rangle$, we obtain the combined system $C$, which is presented in Fig. 10. This system is associate with the justice require-

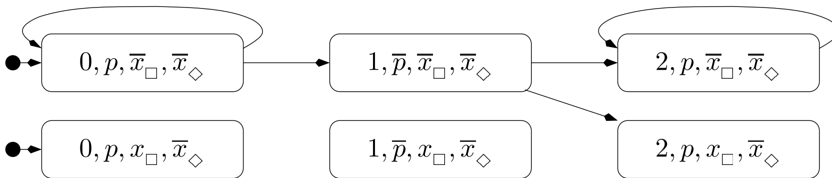
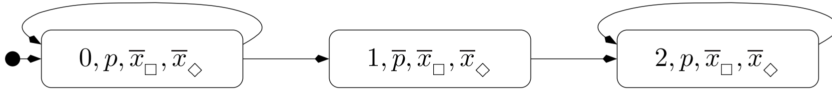



**Figure 10.** Combined System $C$

ments $\neg p \vee x_\square$ and $x_\square \vee \neg x_\diamondsuit$.

Eliminating all unreachable states and states with no successors, we are left with:




State 2 is eliminated because it does not have a path leading to a $(\neg p \vee x_\Box)$-state. Then state 1 is eliminated. having no successors. Finally, 0 is eliminated because it cannot reach a $(\neg p \vee x_\Box)$-state. Nothing is left, hence the system satisfies the property $\Diamond\Box\, p$.

*Correctness of the Algorithms*

**Claim 9**
*For an* FDS $\mathcal{D}$ *and temporal formula* $\varphi$, $\mathcal{D} \models \varphi$ iff $C : \mathcal{D} \; ||| \; T[\varphi] \; ||| \; \langle \Theta : \neg x_\varphi \cdots \rangle$ *is infeasible*

*Proof:*
The proof is based on the observation that every computation of the combined system $C$ is a computation of $\mathcal{D}$ which satisfies the negation of $\varphi$. Therefore, the existence of such a computation shows that not all computations of $\mathcal{D}$ satisfy $\varphi$, and therefore, $\varphi$ is not valid over $\mathcal{D}$. ▟

## 6. Controller Synthesis

In this section we consider the situation that we are given a transition system $\mathcal{D}$ and wish to design for it a controller that will guarantee that all possible executions will satisfy a given LTL specification $\varphi$.

This formulation of the problem is inspired by the classical continuous-time controller synthesis problem. Consider a system such as the one presented in Fig. 11.

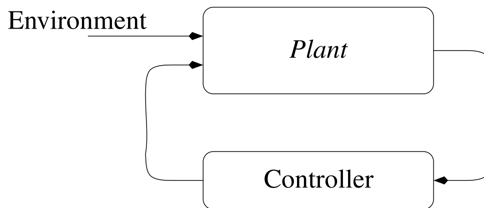



**Figure 11.** A continuous plant and its controller

Such a system typically consists of a physical plant which is controlled by an external controller, which has limited capabilities for observing the internal state of the plant, and influencing its behavior.

Required: A design for a controller which will cause the *plant* to behave correctly under all possible (appropriately constrained) environments.

Due to the limited ability of the controller to observe and manipulate the plant, this problem cannot always be solved. The central problem of control theory is to recognize the cases in which the problem can be solved, and in such cases, produce a design for a controller that can achieve the desired objective.

*Discrete Event Systems Controller*

In [18], Ramadge and Wonham consider the application of this paradigm to the design of controllers for discrete event systems, which is a simplistic model for reactive systems.

They assume a given *plant* which describes the possible events and actions. Some of the actions are controllable while the others are uncontrollable.

Required: Finding a *strategy* for the controllable actions which will maintain a correct behavior against all possible adversary moves. The strategy is obtained by pruning some controllable transitions.

This preliminary work has been later followed by subsequent application of the same approach to the synthesis of *reactive modules*.

In [15] and [1], it is assumed that the *Plant* represents all possible actions. Module actions are controllable. Environment actions are uncontrollable.

Required: Find a *strategy* for the controllable actions which will maintain a temporal specification against all possible adversary moves. Derive a program from this strategy. The problem can be viewed as a *two-persons game*.

We proceed to show how the *controller synthesis* paradigm can be applied to the synthesis of reactive programs (or designs, in general).

### 6.1. Example Design: Arbiter

Consider a specification for an *arbiter* whose architecture layout is presented in Fig. 12.

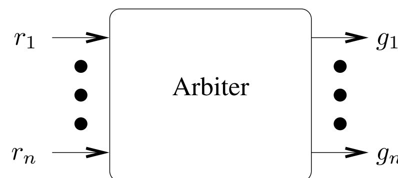



**Figure 12.** Architecture of an Arbiter

The arbiter is expected to allocate a single resource among $n$ clients. The clients post their requests for the resource on the input signals $r_1, \ldots, r_n$ and receive notification of their grant on the arbiter's output signals $g_1, \ldots, g_n$.

The protocol of communication between each client and the arbiter follows the cyclic behavior described in Fig. 13.

Thus, the initial state is when both $r_i$ and $g_i$ are low (0). Then, the client acts first by setting $r_i$ to high (1). This signals a request to access the shared resource. Next, it is the turn of the system (Arbiter) to respond by raising the *grant* signal $g_i$ to *high* (1). Sometimes later, the client is ready to relinquish the resource and this is signaled by lowering $r_i$ to 0. The Arbiter acknowledges the release of the resource by resetting the grant signal $g_i$ to 0. In this diagram we introduce the graphical convention by which uncontrollable actions, performed by the environment are drawn as solid arrows, while controllable actions which are performed by the controller (system) are drawn as dashed arrows (in fact dash-dot font).

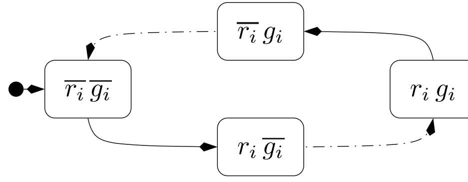



**Figure 13.** Communication between the Arbiter and Client $i$

The overall specification of the system is given by the LTL formula

$$\bigwedge_{i \neq j} \Box \neg(g_i \wedge g_j) \qquad \wedge \qquad \bigwedge_{i} \Box \Diamond (g_i = r_i)$$

The first conjunct expresses the safety property of exclusion, by which at most one *grant* signal may be high at any execution state. The second conjunct expresses the liveness (response) property by which the Arbiter eventually responds to any request made by any client. By requiring that infinitely often $g_i = r_i$, we cover the case of a request for the resource eventually been granted, as well as an eventual acknowledgment of any release of the resource.

*Start by Controller Synthesis*

Assume a given *platform* (plant), identifying the controllable (system) and uncontrollable (environment) transitions for the Arbiter System. This comprehensive transition system is presented in Fig. 14

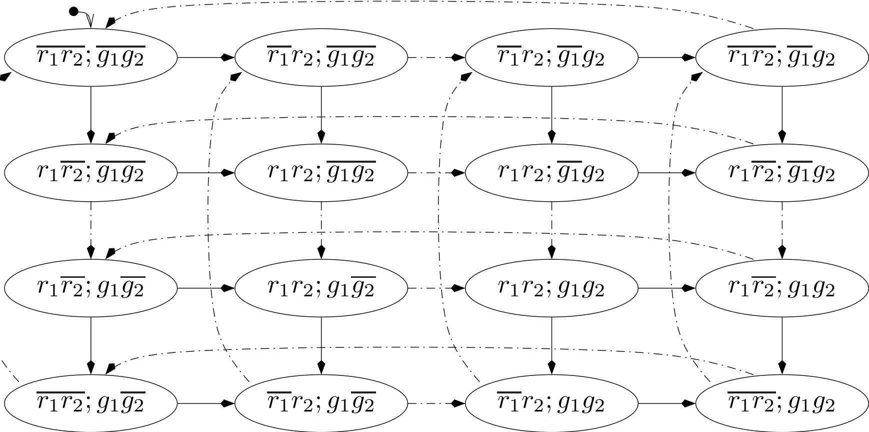


**Figure 14.** Complete transition system for the Arbiter

Due to idling, every node is connected to itself by both dashed and solid transitions, which for simplicity we do not explicitly draw. A complete move consists of a solid edge followed by a dashed edge. Also given is an LTL specification (winning condition):

$$\varphi : \quad \Box \neg(g_1 \wedge g_2) \ \wedge \ \Box \Diamond (g_1 = r_1) \ \wedge \ \Box \Diamond (g_2 = r_2)$$

For Simplicity, we added to the specification the requirement that, at every complete move, *at most one* of the four variables $r_1, r_2, g_1, g_2$ may change its value.

*6.2. Controller Synthesis Via Game Playing*

A *game* is given by $\mathcal{G} : \langle V = \vec{x} \cup \vec{y}, \Theta_1, \Theta_2, \rho_1, \rho_2, \varphi\rangle$, where

- $V = \vec{x} \cup \vec{y}$ are the *state variables*, with $\vec{x}$ being the environment's (player 1) variables, and $\vec{y}$ being the system's (player 2) variables. A state of the game is an interpretation of $V$. Let $\Sigma$ denote the set of all states.
- $\Theta_1(\vec{x})$ — the *initial condition* for player 1 (Environment). An assertion characterizing the environment's initial states.
- $\Theta_2(\vec{x}, \vec{y})$ — the *initial condition* for player 2 (system).
- $\rho_1(\vec{x}, \vec{y}, \vec{x}')$ — *Transition relation* for player 1 (Environment).
- $\rho_2(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$ — *Transition relation* for player 2 (system).
- $\varphi$ — The *winning condition*. An LTL formula characterizing the plays which are winning for player 2.

A state $s_2$ is said to be a $\mathcal{G}$*-successor* of state $s_1$, if both $\rho_1(s_1[V], s_2[\vec{x}])$ and $\rho_2(s_1[V], s_2[V])$ are true.
We denote by $D_{\vec{x}}$ and $D_{\vec{y}}$ the domains of variables $\vec{x}$ and $\vec{y}$, respectively.

*Plays and Strategies*

Let $\mathcal{G} : \langle V, \Theta_1, \Theta_2, \rho_1, \rho_2, \varphi\rangle$ be a game. A *play* of $\mathcal{G}$ is an infinite sequence of states

$$\pi : \quad s_0, s_1, s_2, \ldots,$$

satisfying the requirement:

- Consecution: For each $j \geq 0$, the state $s_{j+1}$ is a $\mathcal{G}$-successor of the state $s_j$.

A play $\pi$ is said to be winning for player 2 if $\pi \models \varphi$. Otherwise, it is said to be winning for player 1.

A *strategy* for player 1 is a function $\sigma_1 : \Sigma^+ \mapsto D_{\vec{x}}$, which determines the next set of values for $\vec{x}$ following any history $h \in \Sigma^+$. A play $\pi : s_0, s_1, \ldots$ is said to be *compatible* with strategy $\sigma_1$ if, for every $j \geq 0$, $s_{j+1}[\vec{x}] = \sigma_1(s_0, \ldots, s_j)$.

Strategy $\sigma_1$ is *winning* for player 1 from state $s$ if all $s$-originated plays (i.e., plays $\pi : s = s_0, s_1, \ldots$) compatible with $\sigma_1$ are winning for player 1. If such a winning strategy exists, we call $s$ a *winning state* for player 1.

Similar definitions hold for player 2 with strategies of the form $\sigma_2 : \Sigma^+ \times D_{\vec{x}} \mapsto D_{\vec{y}}$.

*From Winning Games to Programs*

A game $\mathcal{G}$ is said to be winning for player 2 if every $\vec{x}$-interpretation $\xi$ satisfying $\Theta_1(\xi) = 1$ can be matched by a $\vec{y}$-interpretation $\eta$ satisfying $\Theta_2(\xi, \eta) = 1$, such that the state $\langle \vec{x} : \xi,\ \vec{y} : \eta \rangle$ is winning for 2.

Otherwise, i.e., there exists an interpretation $\vec{x} : \xi$ satisfying $\Theta_1(\xi)$ such that, for all interpretations $\vec{y} : \eta$ satisfying $\Theta_2(\xi, \eta)$, the state $\langle \vec{x} : \xi,\ \vec{y} : \eta \rangle$ is winning for 1, the game is winning for player 1.

We solve the game, attempting to decide whether the game is winning for player 1 or 2. If it is winning for player 1 the specification is unrealizable. If it is winning for player 2, we can extract a winning strategy which is a working implementation.
When applying *controller synthesis*, the platform provides the transition relations $\rho_1$ and $\rho_2$, as well as the initial condition.
Thus, the essence of synthesis under the *controller* framework is an algorithm for computing the set of winning states for a given platform and specification $\varphi$.

*The Game for the Sample Specification*

For the sample specification in which the client-server protocol is given by Fig. 13 and was required to satisfy

$$\bigwedge_{i \neq j} \Box \neg(g_i \wedge g_j) \qquad \wedge \qquad \bigwedge_{i} \Box \Diamond (g_i = r_i)$$

We take the following game structure:

$$\begin{array}{ll}
\vec{x} \cup \vec{y} : \{r_i \mid i = 1, \ldots, n\} \cup \{g_i \mid i = 1, \ldots, n\} & \\
\Theta_1 : \bigwedge_i \overline{r_i} & \Theta_2 : \quad \bigwedge_i \overline{g_i} \\
\rho_1 : \bigwedge_i ((r_i \neq g_i) \rightarrow (r_i' = r_i)) & \\
\rho_2 : \bigwedge_i ((r_i = g_i) \rightarrow (g_i' = g_i)) & \\
\varphi : \bigwedge_{i \neq j} \Box \neg(g_i \wedge g_j) \ \wedge \ \bigwedge_i \Box \Diamond (g_i = r_i) &
\end{array}$$

Note that the safety parts of the specification as implied by the protocol are placed in the local components ($\Theta_1, \Theta_2, \rho_1, \rho_2$), while the requirement of exclusion and the *liveness* part are relegated to the winning condition.

*The Controlled Predecessor*

As in symbolic model checking, computing the winning states involves fix-point computations over a basic predecessor operator. For model checking the operator is $\mathbf{E} \bigcirc p$ satisfied by all states which have a $p$-state as a successor.
For synthesis, we use the *controlled predecessor* operator $\diamondsuit\!\!\!\!\bigcirc\, p$. Its semantics can be defined by

$$\diamondsuit\!\!\!\!\bigcirc\, p : \quad \forall \vec{x}' : \rho_1(V, \vec{x}') \rightarrow \exists \vec{y}' : \rho_2(V, V') \wedge p(V')$$

where $\rho_1$ and $\rho_2$ are the transition relations of the environment and system, respectively.

In our graphic notation, $s \models \diamondsuit\, p$ iff $s$ has at least one dashed $p$-successor, and all solid successors different from $s$ satisfy $p$.

*Solving* $\square\, p$ *Games, Iteration* 0

The set of winning states for a specification $\square\, p$ can be computed by the fix-point expression:

$$\nu Y.\, p \wedge \diamondsuit Y \quad = \quad 1 \wedge p \wedge \diamondsuit p \wedge \diamondsuit\diamondsuit p \wedge \cdots$$

We illustrate this on the specification $\square\, \neg(g_1 \wedge g_2)$. In Fig. 15, we present iteration 0 of the fix-point expression which yields $Y_0 : 1$, i.e. all reachable states.

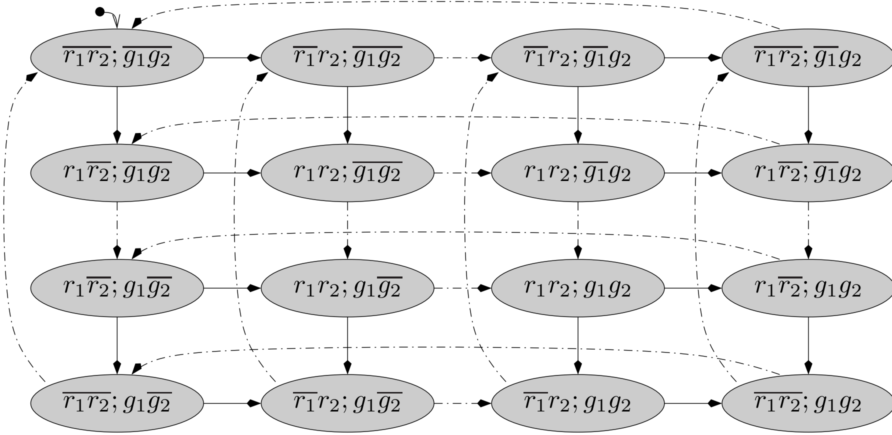



**Figure 15.** Iteration 0, $Y_0 : 1$

Proceeding to the next iteration, we obtain $Y_1 : \neg(g_1 \wedge g_2) \wedge \diamondsuit 1$. The result is presented in Fig. 16. As can be seen in the diagram, this iteration removed the four states which lie at the bottom right corner. These are the states which violate the requirement of exclusion by having $g_1 = g_2 = 1$.
The iteration converges at this point, i.e. $Y_2 = Y_1$. This can be explained by observing that no state which belong to $Y_1$ should be removed in the next iteration. Note that all the four states which have a successor outside of $Y_1$ do not have a solid successor which will take us out of $Y_1$. All the solid edges departing from these four states lead to other $Y_1$-states.

From now on, we can restrict our attention to the transition system presented in Fig. 16 because all winning games must satisfy the exclusion requirement and therefor avoid the four states that have been removed.

*Solving* $\Diamond\, q$ *Games, Iteration* 1

The set of winning states for a specification $\Diamond\, q$ can be computed by the fix-point expression:

$$\mu Y.\, q \vee \diamondsuit Y \quad = \quad q \vee \diamondsuit q \vee \diamondsuit\diamondsuit q \vee \cdots$$

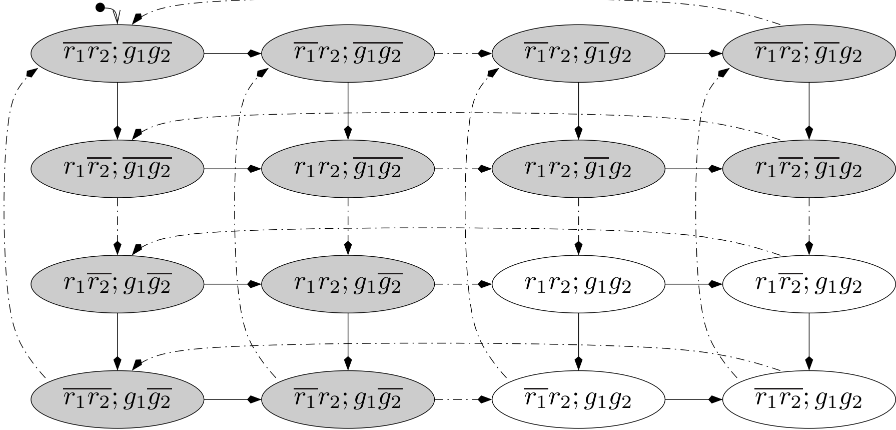



**Figure 16.** Iteration 1, $Y_1 : \neg(g_1 \wedge g_2) \wedge \Diamond\!\!\!\!\bigcirc\, 1$

We illustrate this on the specification $\Diamond(g_1 = r_1)$.

In Fig. 17 we present the first iteration of this fix-point when applied to the transition system of Fig. 16. The iteration yields $Y_1 : (g_1 = r_1)$ identifies as *good states* all the states that satisfy $g_1 = r_1$.

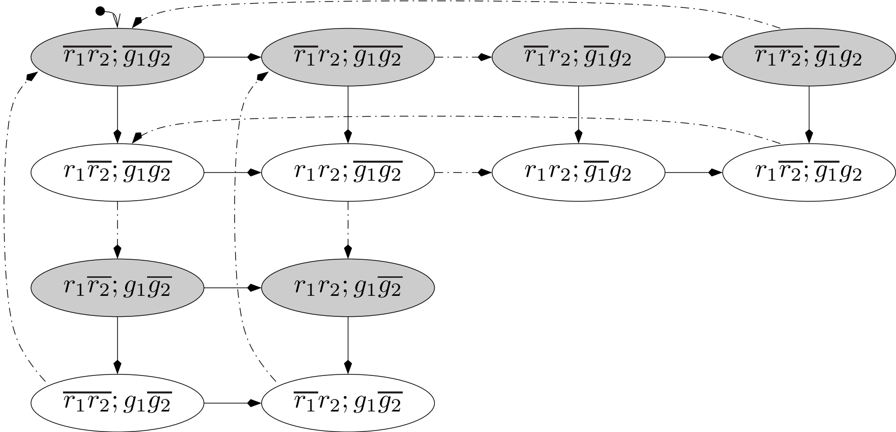



**Figure 17.** Iteration 1, $Y_1 : (g_1 = r_1)$

The next iteration $Y_2 : Y_1 \vee \Diamond\!\!\!\!\bigcirc\, Y_1$ adds to the set of good states the states $(r_1 r_2; \overline{g_1 g_2})$ and $(\overline{r_1} r_2; g_1 \overline{g_2})$. Both of these states have a dashed successor ($(r_1 r_2; g_1 \overline{g_2})$ and $(\overline{r_1} r_2; \overline{g_1 g_2})$, respectively) which is already in $Y_1$, and they do not have any solid successor different than themselves. Iteration $Y_2$ is presented in Fig. 18.

Iteration $Y_3$, presented in Fig. 19 adds to the set of good states the states $(r_1 \overline{r_2}; \overline{g_1 g_2})$ and $(\overline{r_1 r_2}; g_1 \overline{g_2})$. These states have the states $(r_1 \overline{r_2}; g_1 \overline{g_2}), (\overline{r_1 r_2}; \overline{g_1 g_2}) \in Y_2$ as dashed successors, and all of their solid successors are also in $Y_2$.

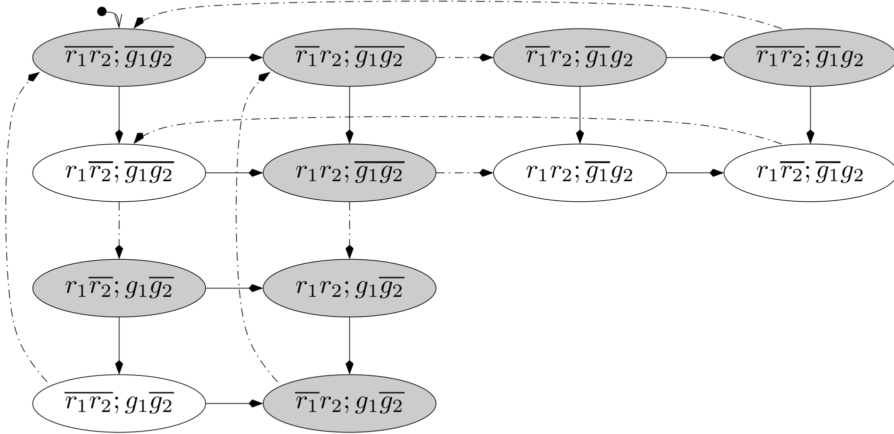



**Figure 18.** Iteration 2, $Y_2 : Y_1 \lor \Diamond Y_1$

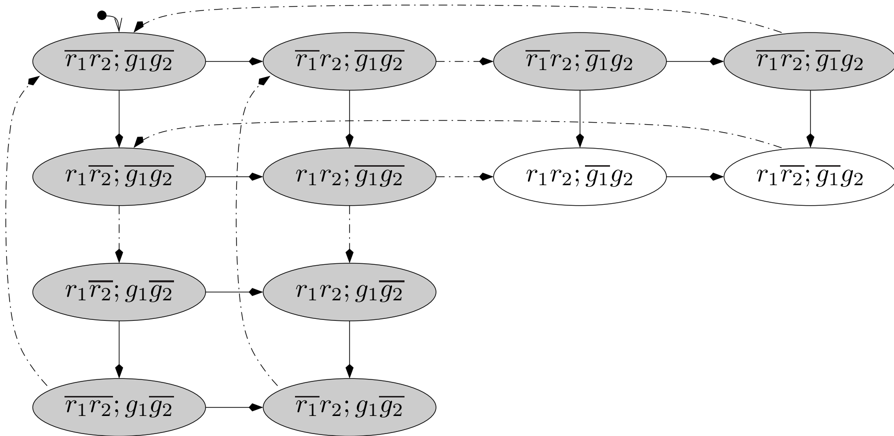



**Figure 19.** Iteration 3, $Y_3 : Y_2 \lor \Diamond Y_2$

Finally, the fourth and last iteration adds to $Y_4$ the single state $(r_1\overline{r_2}; \overline{g_1}g_2)$ which has $(r_1\overline{r_2}; g_1\overline{g_2}) \in Y_3$ as a dashed successor and no external solid successor. This final (and convergent) iteration is presented in Fig. 20.

Note that state $(r_1 r_2; \overline{g_1}g_2)$ is not included in the set of sinning states. This implies that, starting at this state, the Arbiter cannot force the combined system into a state at which $g_1 = r_1$. This is a state in which Client $C_1$ has requested the resource, but a grant is not guaranteed. Why is that? Because currently Client $C_2$ holds the resource and there is no obligation on its behalf to ever release it. Until Client $C_2$ releases the resource, there is no way that the Arbiter can grant the resource to $C_1$ without violating the requirement of exclusion.

This indicates that the original specification may be unrealizable without imposing additional obligations on the behavior of the clients.

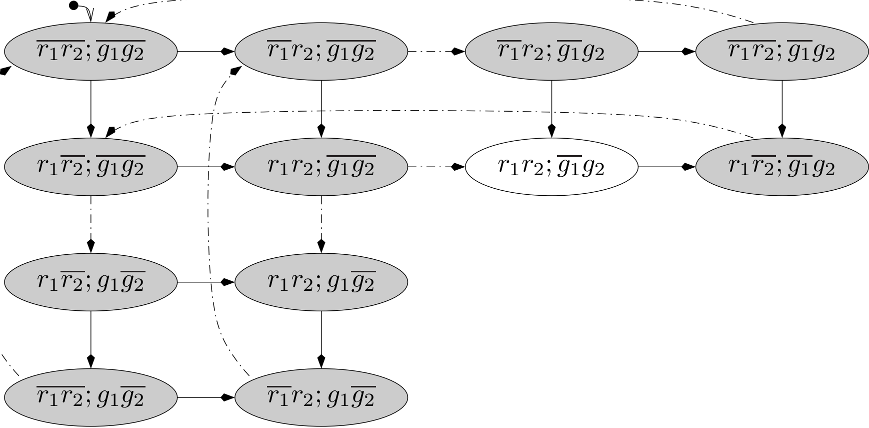



**Figure 20.** Iteration 4, $Y_4 : Y_3 \lor \Diamond\kern-0.65em\bigcirc Y_3$

*Solving* $\Box\Diamond q$ *Games*

A game for a winning condition of the form $\Box\Diamond q$ can be solved by the fix-point expression:

$$\nu Z \mu Y.\, q \land \Diamond\kern-0.65em\bigcirc Z \;\lor\; \Diamond\kern-0.65em\bigcirc Y$$

This is based on the maximal fix-point solution of the equation

$$Z \quad = \quad \mu Y.\,(q \land \Diamond\kern-0.65em\bigcirc Z) \;\lor\; \Diamond\kern-0.65em\bigcirc Y$$

This nested fix-point computation can be computed iteratively by the program:

$$\begin{array}{l} Z := 1 \\ \textbf{Fix}\ (Z) \\ \quad \left[\begin{array}{l} G := q \ \land\ \Diamond\kern-0.65em\bigcirc Z \\ Y := 0 \\ \textbf{Fix}\ (Y) \\ \quad [Y := G \ \lor\ \Diamond\kern-0.65em\bigcirc Y] \\ Z := Y \end{array}\right] \end{array}$$

*Solving* $\Box\Diamond(g_1 = r_1)$ *for the Arbiter Example*

Applying the above fix-point iterations to the Arbiter example, we obtain the set of winning stated depicted in Fig. 21.

Note that the obtained strategy, keeps $g_2 = 0$ permanently. This suggests that we will have difficulties finding a solution that will maintain

$$\Box\Diamond(g_1 = r_1) \ \land\ \Box\Diamond(g_2 = r_2)$$

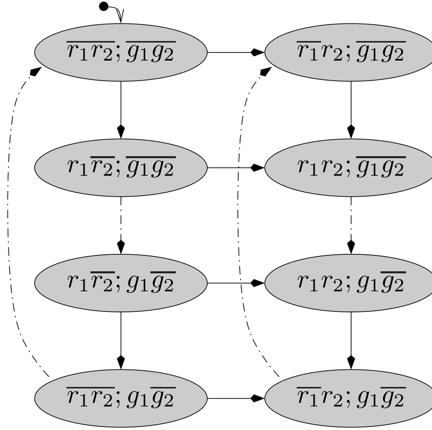



**Figure 21.** Set of winning states for the winning condition $\square\diamondsuit)g_1 = r_1$

*Generalized Response (Büchi)*

In order to solve the game for a winning condition of the form $\square\diamondsuit\, q_1 \wedge \cdots \wedge \square\diamondsuit\, q_n$, we may use the following vector-form fix-point expression:

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \\ \vdots \\ \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y\left((q_1 \wedge \bigcirc\!\!\!\!\diamond\, Z_2) \;\vee\; \bigcirc\!\!\!\!\diamond\, Y\right) \\ \mu Y\left((q_2 \wedge \bigcirc\!\!\!\!\diamond\, Z_3) \;\vee\; \bigcirc\!\!\!\!\diamond\, Y\right) \\ \vdots \\ \vdots \\ \mu Y\left((q_n \wedge \bigcirc\!\!\!\!\diamond\, Z_1) \;\vee\; \bigcirc\!\!\!\!\diamond\, Y\right) \end{bmatrix}$$

Iteratively:

$$\begin{array}{l}
\textbf{For } (i \in 1..n) \textbf{ do } [Z[i] := 1] \\
\textbf{Fix } (Z[1]) \\
\quad \left[\begin{array}{l}
\textbf{For } (i \in 1..n) \textbf{ do} \\
\quad \left[\begin{array}{l}
Y := 0 \\
\textbf{Fix } (Y) \\
\quad [Y := (q[i] \wedge \bigcirc\!\!\!\!\diamond\, Z[i \oplus_n 1]) \;\vee\; \bigcirc\!\!\!\!\diamond\, Y] \\
Z[i] := Y
\end{array}\right]
\end{array}\right] \\
\textbf{Return } Z[1]
\end{array}$$

*Specification is Unrealizable*

Applying the above algorithm to the specification

$$\square\,\diamondsuit(g_1 = r_1) \;\wedge\; \square\,\diamondsuit(g_2 = r_2)$$

we find that it fails. Conclusion:

*The considered specification is unrealizable*

Indeed, without an environment obligation of releasing the resource once it has been granted, the arbiter cannot satisfy any other client.

*A Realizable Specification*

Consider a specification consisting of the protocol as specified in the diagram of Fig. 13. and the temporal specification

$$\bigwedge_{i \neq j} \square \neg(g_i \wedge g_j) \ \wedge \ \left( \bigwedge_i \square \diamondsuit \neg(r_i \wedge g_i) \quad \rightarrow \quad \bigwedge_i \square \diamondsuit (g_i = r_i) \right)$$

We take the following game components:

$$\begin{array}{rl} \vec{x} \cup \vec{y} : & \{r_i \mid i = 1, \ldots, n\} \cup \{g_i \mid i = 1, \ldots, n\} \\ \Theta_1 : & \bigwedge_i \overline{r_i} \qquad\qquad\qquad\qquad \Theta_2 : \quad \bigwedge_i \overline{g_i} \\ \rho_1 : & \bigwedge_i ((r_i \neq g_i) \rightarrow (r_i' = r_i)) \\ \rho_2 : & \bigwedge_{i \neq j} \neg(g_i' \ \wedge \ g_j') \ \wedge \ \bigwedge_i ((r_i = g_i) \rightarrow (g_i' = g_i)) \\ \varphi : & \bigwedge_i \square \diamondsuit \neg(r_i \wedge g_i) \quad \rightarrow \quad \bigwedge_i \square \diamondsuit (g_i = r_i) \end{array}$$

Note that, in this formulation, the safety property of exclusion has been incorporated as part of the transition relation of the Arbiter.

*Solving in Polynomial Time a Doubly Exponential Problem*

The paper [16] provided a general solution to the problem of program synthesis from an LTL specification. It showed that any approach that starts with the standard translation from LTL to Büchi automata, has a doubly exponential lower bound. The first exponent comes from the translation of an LTL formula into a non-deterministic Büchi automaton. The second exponent is due to the determinization of the Büchi automaton into a deterministic Rabin automaton.

One of the messages resulting from the work reported here is

> *Do not be too hasty to translate* LTL *into automata. Try first to locate the formula within the temporal hierarchy, as presented in Fig. 2*

For each class of formulas, synthesis can be performed in polynomial time.

*Solving Games for Generalized Reactivity[1] (Streett[1])*

Following [KPP03], we present an $n^3$ algorithm for solving games whose winning condition is given by the (generalized) Reactivity[1] (GR(1)) condition

$$(\square\diamondsuit p_1 \ \wedge \ \square\diamondsuit p_2 \ \wedge \ \cdots \ \wedge \ \square\diamondsuit p_m) \ \rightarrow \ \square\diamondsuit q_1 \ \wedge \ \square\diamondsuit q_2 \ \wedge \ \cdots \ \wedge \ \square\diamondsuit q_n$$

This class of properties is bigger than the properties specifiable by deterministic Büchi automata. It covers a great majority of the properties we have seen so far.
For example, it covers the realizable version of the specification for the Arbiter design.

*The Solution*

The winning states in a React[1] game can be computed by

$$\varphi = \nu \begin{bmatrix} Z_1 \\ \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y \left( \bigvee_{j=1}^{m} \nu X (q_1 \wedge \Diamond Z_2 \ \vee \ \Diamond Y \ \vee \ \neg p_j \wedge \Diamond X) \right) \\ \mu Y \left( \bigvee_{j=1}^{m} \nu X (q_2 \wedge \Diamond Z_3 \ \vee \ \Diamond Y \ \vee \ \neg p_j \wedge \Diamond X) \right) \\ \vdots \\ \vdots \\ \mu Y \left( \bigvee_{j=1}^{m} \nu X (q_n \wedge \Diamond Z_1 \ \vee \ \Diamond Y \ \vee \ \neg p_j \wedge \Diamond X) \right) \end{bmatrix}$$

where

$$\Diamond \varphi : \quad \forall \vec{x}' : \rho_1(V, \vec{x}') \rightarrow \exists \vec{y}' : \rho_2(V, V') \wedge \varphi(V')$$

*Results of Synthesis*

The design realizing the specification can be extracted as the winning strategy for Player *2*. Applying this to the Arbiter specification, we obtain the design presented in Fig. 22.

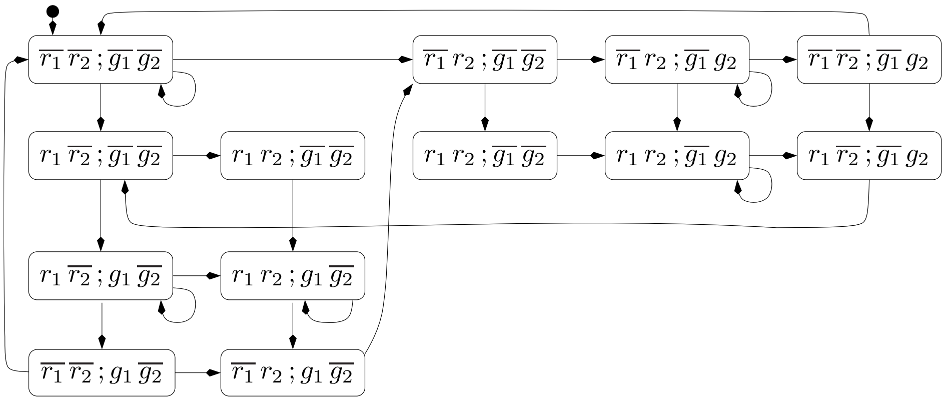



**Figure 22.** Resulting design for Arbiter with $n = 2$

There exists a symbolic algorithm for extracting the implementing design/winning strategy.

*Execution Times and Programs Size for Arbiter(n)*

In Fig. 23, we present a graph which displays the time and space for computing the implementing design for various values of $n$ which counts the number of clients.

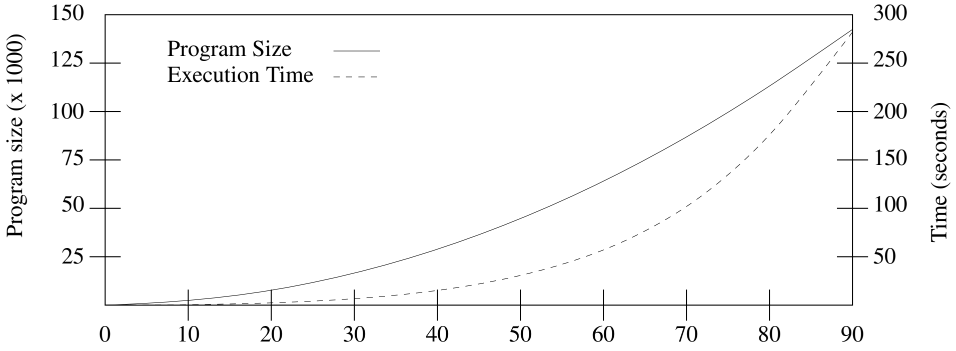



**Figure 23.** Execution time and space for synthesizing design for Arbiter($n$)

## 7. Extraction of Designs

It remains to show how to extract a *winning strategy* for the case that a game is winning for player 2.
Let $\mathcal{G} : \langle V = \vec{x} \cup \vec{y}, \Theta_1, \Theta_2, \rho_1, \rho_2, \varphi\rangle$ be a given game. A *controller for* $\mathcal{G}$ is an FDS $\mathcal{G}_c : \langle V_c, \Theta_c, \rho_c, \emptyset, \emptyset\rangle$, such that:

- $V_c \supseteq V$. That is, $V_c$ extends the set of variables of $\mathcal{G}$.
- $\Theta_c \rightarrow \Theta_1 \wedge \Theta_2$. That is, every $\mathcal{G}_c$-initial state satisfies both $\Theta_1$ and $\Theta_2$.
- $\rho_c \rightarrow \rho$, where $\rho = \rho_1 \wedge \rho_2$. That is, if $s_2$ is a $\rho_c$-successor of $s_1$, then $s_2$ is also a $(\rho_1 \wedge \rho_2)$-successor of $s_1$.
- *Player-1 Completeness* — $\Theta_c\Downarrow_{\vec{x}} = \Theta_1$ and $\rho_c\Downarrow_{V,\vec{x}'} = \rho_1$. That is, when projecting the initial states of $\mathcal{G}_c$ on the variables $\vec{x}$, we obtain precisely $\Theta_1$. Also, a state $s_1 \in \Sigma_c$ has a $\rho_1$-successor $s_2$ iff $s_1$ has a $\rho_c$-successor which agrees with $s_2$ on the valuation of $\vec{x}$.
- Every infinite run of $\mathcal{G}_c$ satisfies the winning condition $\varphi$.

*Example: Extracted Controller for Arbiter*

In Fig. 24, we present again the controller extracted for the Arbiter example.

*Interpreting a Controller as a Program*

A program (equivalently, a circuit) implementing the extracted controller follows the states that are contained in $\mathcal{G}_c$. It has a program counter which ranges over the states of $\mathcal{G}_c$.
Assume that control is currently at state $S$ of $\mathcal{G}_c$. Let the next values of the input variables be $\vec{x} = \xi$. Choose a state $S'$ which is a $\rho_c$-successor of $S$, and such that $S'[\vec{x}] = \xi$. By the requirement of *Player-1 Completeness*, there always exists such a successor.
The actions of the program is to output the values $\eta$ such that $S'[\vec{y}] = \eta$, and to move to state $S'$.

In the following sequence of slides, we present for various winning conditions the algorithm for computing the set of winning states and an algorithm for extracting a controller in the case the game is winning for player 2.

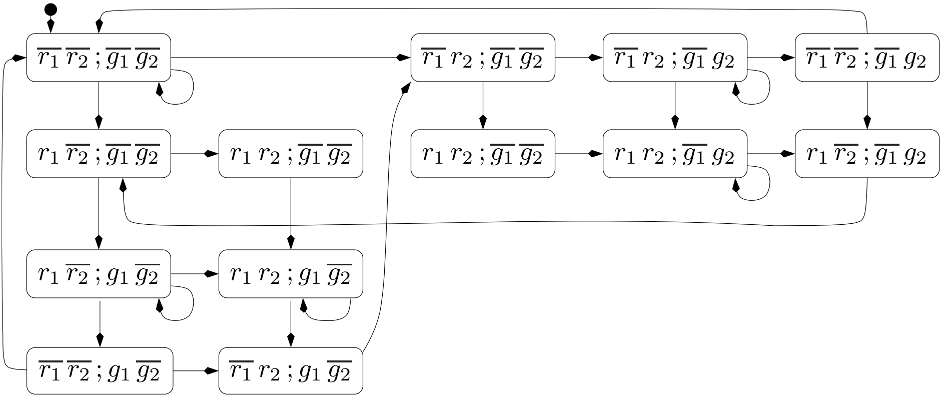



**Figure 24.** A program for an Arbiter for 2 Clients

*Computing a Controller for the Winning Condition* $\square\, p$

The winning states in a game with a winning condition $\square\, p$ are given by:

$$win \quad = \quad \nu Z.\, p \;\wedge\; \circledcirc\, Z$$

The full controller extraction algorithm can be given by the following program:

$$\begin{array}{l} Z := 1 \\ \textbf{Fix}\ (Z) \\ \quad \big[ Z := p \wedge \circledcirc\, Z \big] \\ \textbf{if}\ (\Theta_1 \wedge \neg(\exists \vec{y} : \Theta_2 \wedge Z)) \neq 0\ \textbf{then} \\ \quad \textbf{Print "Specification is unrealizable"} \\ \textbf{else} \\ \quad \begin{bmatrix} \Theta_c := \Theta_1 \wedge \Theta_2 \wedge Z \\ \rho_c \;\; := Z \wedge \rho \wedge Z' \end{bmatrix} \end{array}$$

**Claim 10** *If $s$ is a winning state of a $(\square\, p)$-game, then $s \models p$, and player 2 can force the game to move from $s$ to a successor which is also a winning state.*

*Computing A Controller for the Winning Condition* $\diamondsuit\, q$

The winning states in a game with a winning condition $\diamondsuit\, q$ are given by:

$$win \quad = \quad \mu Y.\, q \;\vee\; \circledcirc\, Y$$

The full controller extraction algorithm can be given by the following program:

$$
\begin{array}{l}
Y := q; \quad r := 0; \quad U[0] := q \\
\textbf{Fix}\ (Y) \\
\quad \left[ Y := q \vee \bigotimes Y; \quad r := r+1; \quad U[r] := Y \right] \\
\textbf{if}\ (\Theta_1 \wedge \neg(\exists \vec{y} : \Theta_2 \wedge Y)) \neq 0\ \textbf{then} \\
\quad \textbf{Print "Specification is unrealizable"} \\
\textbf{else} \\
\quad \left[ \begin{array}{l}
\Theta_c := \Theta_1 \wedge \Theta_2 \wedge Y \\
\rho_c := 0; \quad \mathit{prev} := U[0] \\
\textbf{for}\ i \in 1 \ldots r\ \textbf{do} \\
\quad \left[ \rho_c := \rho_c \ \vee \ (U[i] \wedge \neg \mathit{prev}) \wedge \rho \wedge \mathit{prev}'; \quad \mathit{prev} := \mathit{prev} \vee U[i] \right]
\end{array} \right]
\end{array}
$$

**Claim 11** *Every winning state $s$ in a $(\diamondsuit q)$-game is associated with a natural rank $r(s) \geq 0$, such that if $r(s) = 0$ then $s \models q$, and if $r(s) > 0$, then player 2 can force the game to move from $s$ to a winning successor with a lower rank.*

*Computing A Controller for the Winning Condition $\square \diamondsuit q$*

The winning states in a game with a winning condition $\square \diamondsuit q$ are given by:

$$
\mathit{win} \quad = \quad \nu Z \mu Y. (q \wedge \bigotimes Z) \ \vee \ \bigotimes Y
$$

The full controller extraction algorithm can be given by the following program:

$$
\begin{array}{l}
Z := 1 \\
\textbf{Fix}\ (Z) \\
\quad \left[ \begin{array}{l}
Y := q \wedge \bigotimes Z; \quad r := 0; \quad U[0] := Y \\
\textbf{Fix}\ (Y) \\
\quad \left[ Y := (q \wedge \bigotimes Z) \vee \bigotimes Y; \quad r := r+1; \quad U[r] := Y \right] \\
Z := Y
\end{array} \right] \\
\textbf{if}\ (\Theta_1 \wedge \neg(\exists \vec{y} : \Theta_2 \wedge Z)) \neq 0\ \textbf{then Print "Specification is unrealizable"} \\
\textbf{else} \\
\quad \left[ \begin{array}{l}
\Theta_c := \Theta_1 \wedge \Theta_2 \wedge Z \\
\rho_c := U[0] \wedge \rho \wedge Z'; \quad \mathit{prev} := U[0] \\
\textbf{for}\ i \in 1 \ldots r\ \textbf{do} \\
\quad \left[ \rho_c := \rho_c \ \vee \ (U[i] \wedge \neg \mathit{prev}) \wedge \rho \wedge \mathit{prev}'; \quad \mathit{prev} := \mathit{prev} \vee U[i] \right]
\end{array} \right]
\end{array}
$$

**Claim 12** *Every winning state $s$ in a $(\square \diamondsuit q)$-game is associated with a natural rank $r(s)$, such that player 2 can force the game to move from $s$ to a winning successor $s'$ where either $r(s) = 0$ and $s \models q$, or $r(s) > r(s')$.*

*Winning States for the Condition $\square \diamondsuit q_1 \wedge \cdots \wedge \square \diamondsuit q_n$*

The winning states in a game with a winning condition $\square \diamondsuit q_1 \wedge \cdots \wedge \square \diamondsuit q_n$ (generalized *Büchi* ) are given by:

$$win \;=\; \nu\begin{bmatrix} Z_1 \\ Z_2 \\ \\ \vdots \\ \\ Z_n \end{bmatrix}\begin{bmatrix} \mu Y\,((q_1 \wedge \diamondsuit Z_2) \;\vee\; \diamondsuit Y) \\ \mu Y\,((q_2 \wedge \diamondsuit Z_3) \;\vee\; \diamondsuit Y) \\ \vdots \\ \vdots \\ \mu Y\,((q_n \wedge \diamondsuit Z_1) \;\vee\; \diamondsuit Y) \end{bmatrix}[1]$$

This is a case where the extracted strategy needs to rely on an auxiliary memory. The memory will be represented by an additional variable $ix : 1 \ldots n$.

*Controller Extraction for the Condition* $\square\diamond q_1 \wedge \cdots \wedge \square\diamond q_n$

The controller extraction algorithm can be given by the following program:

$$
\begin{array}{l}
\textbf{For } (j \in 1\ldots n) \textbf{ do } [Z[j] := 1] \\
\textbf{Fix } (Z[1]) \\
\quad\left[\begin{array}{l}
\textbf{For } (j \in 1\ldots n) \textbf{ do} \\
\quad\left[\begin{array}{l}
Y := q[j] \wedge \diamondsuit Z[j \oplus_n 1]; \quad r := 0; \quad U[j,0] := Y \\
\textbf{Fix } (Y) \\
\quad [Y := (q[j] \wedge \diamondsuit Z[j \oplus_n 1]) \;\vee\; \diamondsuit Y; \quad r := r+1; \quad U[j,r] := Y] \\
Z[j] := Y; \quad maxr[j] := r
\end{array}\right]
\end{array}\right] \\
\textbf{if } (\Theta_1 \wedge \neg(\exists \vec{y} : \Theta_2 \wedge Z[1])) \neq 0 \textbf{ then Print "Specification is unrealizable"} \\
\textbf{else} \\
\quad\left[\begin{array}{l}
\Theta_c := \Theta_1 \wedge \Theta_2 \wedge Z[1] \wedge ix = 1; \quad \rho_c := 0 \\
\textbf{For } (j \in 1\ldots n) \textbf{ do} \\
\quad\left[\begin{array}{l}
\rho_c := \rho_c \;\vee\; (ix = j) \wedge U[j,0] \wedge \rho \wedge Z'[j \oplus_n 1] \wedge (ix' = j \oplus_n 1) \\
prev := U[j,0] \\
\textbf{for } r \in 1 \ldots maxr[j] \textbf{ do} \\
\quad\left[\begin{array}{l}
\rho_c := \rho_c \;\vee\; (ix = j) \wedge (U[j,r] \wedge \neg prev) \wedge \rho \wedge prev' \wedge (ix' = j) \\
prev := prev \vee U[j,r]
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}
$$

*A Controller for the Winning Condition* $(\square\diamond p \;\rightarrow\; \square\diamond q)$

The winning states in such a game are given by:

$$win \;=\; \nu Z \mu Y \nu X.\,(q \wedge \diamondsuit Z) \;\vee\; \diamondsuit Y \;\vee\; (\neg p \wedge \diamondsuit X)$$

The full controller extraction algorithm can be given by the following program:

$$
\begin{array}{l}
Z := 1 \\
\textbf{Fix}\ (Z) \\
\quad \left[\begin{array}{l}
Y := 0; \quad r := 0 \\
\textbf{Fix}\ (Y) \\
\quad \left[\begin{array}{l}
X := 1 \\
\textbf{Fix}\ (X) \\
\quad [X := (q \wedge \diamondsuit Z) \vee \diamondsuit Y \vee (\neg p \wedge \diamondsuit X)] \\
Y := X; \quad U[r] := Y; \quad r := r + 1
\end{array}\right] \\
Z := Y; \quad \mathit{maxr} := r - 1
\end{array}\right] \\
\textbf{if}\ (\Theta_1 \wedge \neg(\exists \vec{y} : \Theta_2 \wedge Z)) \neq 0\ \textbf{then Print "Specification is unrealizable"} \\
\textbf{else} \left[\begin{array}{l}
\Theta_c := \Theta_1 \wedge \Theta_2 \wedge Z \\
\rho_c := (q \wedge U[0] \wedge \rho \wedge Z') \vee (\neg p \wedge U[0] \wedge \rho \wedge U'[0]); \quad \mathit{prev} := U[0] \\
\textbf{for}\ r \in 1 \ldots \mathit{maxr}\ \textbf{do} \\
\quad \left[\begin{array}{l}
W := U[r] \wedge \neg \mathit{prev}; \quad \mathit{nprev} := \mathit{prev} \vee U[r] \\
\rho_c := \rho_c \vee W \wedge \rho \wedge \mathit{prev}' \vee \neg p \wedge W \wedge \rho \wedge W'; \quad \mathit{prev} := \mathit{nprev}
\end{array}\right]
\end{array}\right]
\end{array}
$$

**Claim 13** *Every winning state $s$ in a $(\Box\Diamond q)$-game is associated with a natural rank $r(s)$, such that player 2 can force the game to move from $s$ to a winning successor $s'$ where either $r(s) = 0$ and $s \models q$, or $r(s) > r(s')$.*

## 8. Synthesis from LTL Specifications

We will now consider synthesis of programs/designs directly from an LTL specification without the intermediary of a *platform* as part of the specification.

*Property-Based System Design*

While the rest of the world seems to be moving in the direction of *model-based* design (see System-C, UML), some of us persist with the vision of property-based approach.

Specification is stated declaratively as a set of *properties*, from which a design can be extracted.

This is currently studied in the project PROSYD.

Design synthesis is needed in two places in the development flow:

- Automatic synthesis of small blocks whose time and space efficiency are not critical.
- As part of the specification analysis phase, ascertaining that the specification is realizable.

*A Realizable Specification*

Reconsider the example of the Arbiter, where the interaction between the Arbiter and the clients follows the protocol described in Fig. 13.

*Assumptions* (Constraints on the Environment)

$$A: \quad \bigwedge_i (\overline{r_i} \wedge (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \wedge r_i \wedge g_i \Rightarrow \Diamond \overline{r_i})$$

*Guarantees* (Expectations from System)

$$G: \quad \bigwedge_{i \neq j} \Box \neg(g_i \wedge g_j) \ \wedge \ \bigwedge_i \left( \overline{g_i} \ \wedge \ \begin{pmatrix} r_i = g_i \Rightarrow \bigcirc\, g_i = g_i \ \wedge \\ r_i \wedge \overline{g_i} \ \Rightarrow \Diamond\, g_i \qquad \wedge \\ \overline{r_i} \wedge g_i \ \Rightarrow \Diamond\, \overline{g_i} \end{pmatrix} \right)$$

*Total Specification*

$$\varphi: \qquad A \rightarrow G$$

*Program Synthesis from* LTL *Specification*

Assume a specification given as an LTL formula $\varphi$ plus an identification of the *input variables* $\vec{x}$ and *output variables* $\vec{y}$. We construct the (structurally trivial) game

$$G_\varphi: \quad \langle V = \vec{x} \cup \vec{y},\ \Theta_1 : 1,\ \Theta_2 : 1,\ \rho_1 : 1,\ \rho_2 : 1,\ \varphi \rangle$$

Solving the game $G_\varphi$ by the *controller synthesis* methods, and extracting the implementing controller, we obtain a program (or circuit) which realizes the specification $\varphi$.

There are several technical considerations which facilitates the application of this approach.

For two games $G_1, G_2$ we say that $G_1$ is *equi-realizable* to $G_2$, written $G_1 \sim G_2$, if the realizability status of $G_1$ is equal to that of $G_2$. That is, if both are realizable, or both are unrealizable.

*GR(1) Specifications over Past Formulas*

There are many cases in which the specification is given as, or is equivalent to, an LTL formula of the form

$$\varphi: \quad (\Box\Diamond\, p_1 \ \wedge \ \cdots \ \wedge \ \Box\Diamond\, p_m) \quad \rightarrow \quad (\Box\Diamond\, q_1 \ \wedge \ \cdots \ \wedge \ \Box\Diamond\, q_n),$$

where $p_1, \ldots, p_m, q_1, \ldots, q_n$ are *past* formulas, rather than assertions. We construct the testers $T[p_1], \ldots, T[p_m], T[q_1], \ldots, T[q_n]$. Let $b_{p_1}, \ldots, b_{p_m}, b_{q_1}, \ldots, b_{q_n}; \Theta_{p_1}, \ldots, \Theta_{p_m}, \Theta_{q_1}, \ldots, \Theta_{q_n}; \rho_{p_1}, \ldots, \rho_{p_m}, \rho_{q_1}, \ldots, \rho_{q_n}$ be the output variables, initial conditions, and transition relations, respectively, of testers $T[p_1], \ldots, T[p_m], T[q_1], \ldots, T[q_n]$.

The synthesis of specifications such as $\varphi$ uses the reduction implied by:

**Claim 14** *The game* $G : \langle V = \vec{x} \cup \vec{y}, \Theta_1, \Theta_2, \rho_1, \rho_2, \varphi \rangle$ *is equi-realizable to the game* $\widetilde{G} : \langle \widetilde{V}, \widetilde{\Theta}_1, \widetilde{\Theta}_2, \widetilde{\rho}_1, \widetilde{\rho}_2, (\bigwedge_{i=1}^{m} \Box\Diamond\, b_{p_i}) \rightarrow (\bigwedge_{j=1}^{n} \Box\Diamond\, b_{q_j}) \rangle$

where, $\widetilde{V}: \ (\vec{x} \cup \{b_{p_1}, \ldots, b_{p_m}\}) \ \cup \ (\vec{y} \cup \{b_{q_1}, \ldots, b_{q_n}\})$
$\widetilde{\Theta}_1 : \Theta_1 \ \wedge \ \Theta_{p_1} \wedge \cdots \wedge \Theta_{p_m}$
$\widetilde{\Theta}_2 : \Theta_2 \ \wedge \ \Theta_{q_1} \wedge \cdots \wedge \Theta_{q_n}$
$\widetilde{\rho}_1 : \ \rho_1 \ \wedge \ \rho_{p_1} \wedge \cdots \wedge \rho_{p_m}$
$\widetilde{\rho}_2 : \ \rho_2 \ \wedge \ \rho_{q_1} \wedge \cdots \wedge \rho_{q_n}$

*A Special Case: Treatment of the Safety Components*

Many specifications have the following form (e.g., the Arbiter example):

$$\varphi : \quad (I_1(\vec{x}) \wedge \square\, R_1(\vec{x}, \vec{y}, \bigcirc\, \vec{x}) \wedge L_1) \;\rightarrow\; (I_2(\vec{x}, \vec{y}) \wedge \square\, R_2(\vec{x}, \vec{y}, \bigcirc\, \vec{x}, \bigcirc\, \vec{y}) \wedge L_2)$$

Obviously, $I_1 \wedge \square\, R_1$ and $I_2 \wedge \square\, R_2$ are the safety parts of the environment and system, respectively.

The following reduction replaces the safety parts by corresponding $\square\diamondsuit\, p$ conjuncts.

**Claim 15** *The game* $G : \langle V = \vec{x} \cup \vec{y}, \Theta_1, \Theta_2, \rho_1, \rho_2, \varphi\rangle$ *is equi-realizable to the game* $\widetilde{G} : \langle \widetilde{V}, \widetilde{\Theta}_1, \widetilde{\Theta}_2, \widetilde{\rho}_1, \widetilde{\rho}_2, (\square\diamondsuit\, b_1 \wedge L_1) \rightarrow (\square\diamondsuit\, b_2 \wedge L_2)\rangle$.

where, $\widetilde{V} : \; (\vec{x} \cup \{b_1) \,\cup\, (\vec{y} \cup \{b_2)$

$\widetilde{\Theta}_1 : \Theta_1 \;\wedge\; b_1 = I_1$

$\widetilde{\Theta}_2 : \Theta_2 \;\wedge\; b_2 = I_2$

$\widetilde{\rho}_1 : \; \rho_1 \;\wedge\; (b_1' = b_1 \wedge R_1(\vec{x}, \vec{y}, \vec{x}'))$

$\widetilde{\rho}_2 : \; \rho_2 \;\wedge\; (b_2' = b_2 \wedge R_2(\vec{x}, \vec{y}, \vec{x}', \vec{y}'))$

This transformation can be formally justified by the equivalence

$$(I_i \;\wedge\; \square\, R_i(V, \bigcirc\, V)) \quad\sim\quad \square\diamondsuit\, \boxminus(\mathit{first} \wedge I_i \;\vee\; \neg\mathit{first} \wedge R_i(\ominus\, V, V))$$

where, $\mathit{first} = \neg \ominus 1$ characterizes the first position in a sequence.

*The* VMCAI'06 *Transformation*

In the paper [14], we considered specifications of the form

$$\varphi : \quad (I_1 \;\wedge\; \square\, R_1 \;\wedge\; L_1) \;\rightarrow\; (I_2 \;\wedge\; \square\, R_2 \;\wedge\; L_2)$$

and proposed to apply the transformation supported by the following claim, to which we refer as the VMCAI'06 *transformation*.

**Claim 16** *Consider the game* $G : \langle V = \vec{x} \cup \vec{y}, \Theta_1, \Theta_2, \rho_1, \rho_2, \varphi\rangle$ *and its transformed version* $\widetilde{G} : \langle V,\; \Theta_1 \wedge I_1,\; \Theta_2 \wedge I_2,\; \rho_1 \wedge R_1,\; \rho_2 \wedge R_2,\; L_1 \rightarrow L_2\rangle$. *If* $\widetilde{G}$ *is realizable, then so is* $G$.

This transformation was used in order to synthesize the designs for the Arbiter specification for up to 100 clients.

*The* VMCAI'06 *Transformation is Sound but Not Complete*

We show a counter-example specification $\varphi : (\square\, R_1 \wedge L_1) \rightarrow (\square\, R_2 \wedge L_2)$ (due to Marco Roveri), such that the game $G : \langle V : \{x, y\}, 1, 1, 1, 1, \varphi\rangle$ is realizable but the transformed game $\widetilde{G} : \langle V, 1, 1, R_1, R_2, L_1 \rightarrow L_2\rangle$ is not.

The specification is:

$$\varphi : \quad (\Box(x' = 0) \ \wedge \ \Box\Diamond(x = y)) \ \rightarrow \ (\Box(y' = x') \ \wedge \ \Box\Diamond(y = 1))$$

An implementing FDS for this specification is the FDS $\mathcal{D}_{y=1}$ in which the transition relation is $y' = 1$. Namely, an FDS in which $y$ continuously equals 1. Consider any behavior $\sigma$ which is compatible with the constraint $y = 1$. If $x$ is continuously 0 then $\sigma$ violates the requirement $\Box\Diamond(x = y)$. Otherwise, $\sigma$ violates the conjunct $\Box(x' = 0)$. In any case, $\sigma$ violates $\Box(x' = 0) \ \wedge \ \Box\Diamond(x = y)$ and, therefore, satisfies $\varphi$. Thus, system $\mathcal{D}_{y=1}$ maintains $\varphi$ against all behaviors of $x$.

On the other hand, the derived game

$$\widetilde{G} : \quad \langle\{x, y\}, \ 1, \ 1, \ x' = 0, \ y' = x', \ \Box\Diamond(x = y) \rightarrow \Box\Diamond(y = 1)\rangle$$

is unrealizable. This is because in any play of this game the values of $x$ and $y$ are forced by the transition relations to be both 0. Such a behavior necessarily violates the implication $\Box\Diamond(x = y) \ \rightarrow \ \Box\Diamond(y = 1)$, under all possible choices of the second player (which does not really have any choices).

*Conclusions*

The obvious conclusion is that the VMCAI'06 transformation cannot be applied in a complete manner to specifications of the form $(\Box\, R_1 \wedge L_1) \rightarrow (\Box\, R_2 \wedge L_2)$. One may conclude that the transformation is at fault.
An alternative conclusion is that specifications of the form $(\Box\, R_1 \wedge L_1) \rightarrow (\Box\, R_2 \wedge L_2)$ are ill-posed since they allow the safety property $\Box\, R_2$ to depend on the *liveness property* $L_1$. A better-posed version of such a specification would be:

$$(\Box\, R_1 \rightarrow \Box\, R_2) \ \wedge \ (\Box\, R_1 \wedge L_1 \rightarrow L_2)$$

in which the safety property of the system depends only on the safety property of the environment, while the *liveness* property of the system depends on both the safety and *liveness* properties of the environment.
Indeed, if we rewrite Roveri's specification as

$$(\Box(x' = 0) \rightarrow \Box(y' = x')) \ \wedge \ (\Box(x' = 0) \wedge \Box\Diamond(x = y) \ \rightarrow \ \Box\Diamond(y = 1))$$

we obtain a specification which is *unrealizable*, as is the derived game obtained by the VMCAI'06 transformation.

*Any Improvement?*

Therefore, we now consider a specification of the form

$$\varphi : \quad (\Box\, R_1 \rightarrow \Box\, R_2) \ \wedge \ (\Box\, R_1 \wedge L_1 \rightarrow L_2)$$

and propose transforming the game $G : \langle V : \{x, y\}, 1, 1, 1, 1, \varphi\rangle$ into the derived game $\widetilde{G} : \langle V, 1, 1, R_1, R_2, L_1 \rightarrow L_2\rangle$.

Again we can show that this transformation is sound but incomplete. The counter-example (due to Oded Maler) in this case is given by the following specification:

$$\square(\underbrace{x' > x}_{R_1}) \quad \rightarrow \quad \square(\underbrace{y' > y}_{R_2})$$

where $x$ and $y$ range over the domains $[0..10]$ and $[0..5]$, respectively, and we may assume $x = 0$ and $y = 0$ as initial conditions. Note that here, both liveness properties $L_1$ and $L_2$ are taken as trivially 1. We can show that this specification is realizable by the FDS $\mathcal{D}_{y=0}$ which keeps $y$ permanently at 0. This is because no behavior of $x$ can satisfy $\square(x' > x)$ for more than 10 steps. On the other hand, the derived game:

$$\widetilde{G} : \langle \{x, y\},\ x = 0,\ y = 0,\ x' > x,\ y' > y,\ 1\rangle$$

is unrealizable because, in any play, the system gets blocked (deadlocks) within 5 steps, while the environment can survive for 10 steps.

*The Ultimate Recommended Form*

Following the sequence of successive refinements, we finally reach the following recommended form for specifications:

$$\varphi : \quad (I_1 \rightarrow I_2)\ \wedge\ (I_1 \rightarrow \square(\boxminus\, R_1 \rightarrow R_2))\ \wedge\ (I_1 \wedge \square R_1 \wedge L_1\ \rightarrow\ L_2)$$

The second conjunct requires that, for all positions $j \geq 0$ in the sequence, if $I_1$ was true at position 0, and $R_1$ held continuously from position 0 up to $j$, then $R_2$ holds at $j$. For such specifications the VMCAI'06 transformation is both sound and complete, as stated by the following:

**Claim 17** *The game* $G : \langle V, \Theta_1, \Theta_2, \rho_1, \rho_2, \varphi\rangle$ *and its derived version* $\widetilde{G} : \langle V,\ \Theta_1 \wedge I_1,\ \Theta_2 \wedge I_2,\ \rho_1 \wedge R_1,\ \rho_2 \wedge R_2,\ L_1 \rightarrow L_2\rangle$ *are equi-realizable.*

As before, we assume that that the user identifies for the environment the components $I_1$, $\square R_1$, and $L_1$, which stand, respectively, for the initial condition, safety part, and liveness part of the environment's obligations. In a similar way, the components $I_2$, $\square R_2$, and $L_2$ are identified as the guarantees of the system. The only difference from the previous formulation which consisted of the single implication $(I_1 \wedge \square R_1 \wedge L_1) \rightarrow (I_2 \wedge \square R_2 \wedge L_2)$ is that now we propose to structure the total specification in the form presented above as $\varphi$.

*Observation*

The class of (generalized) GR(1) specifications is interesting because it captures systems in which both the environment and the synthesized design can be implemented by FDS's which may contains justice but no compassion requirements.

## 9. Conclusions

In this paper we presented an approach to the automatic synthesis of designs (programs) directly from LTL specifications. Such designs are guaranteed to be correct-by-construction and does not require any further verification.

We show that for a wide fragment of LTL synthesis can be performed in time which is polynomial in the size of the specification.

Some of the more concrete conclusions are:

- It is possible to perform design synthesis for restricted fragments of LTL in acceptable time.
- The tractable fragment (GR(1)) covers most of the properties that appear in standard specifications.
- It is worthwhile to invest an effort in locating the formula within the temporal hierarchy. Solving a game in React(k) has complexity $N^{(2k+1)}$.
- The methodology of property-based system design (Prosyd) is an option worth considering. It is greatly helped by improved algorithms for *synthesis*.

## References

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 1–17. Springer-Verlag, 1989.

[2] R. Alur and S. L. Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

[3] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.

[4] J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[5] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963.

[6] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[8] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. and Cont.*, 200(1):35–61, 2005.

[9] Y. Kesten and A. Pnueli. A Compositional Approach to CTL$^*$ Verification. *Theor. Comp. Sci.*, 331(2–3):397–428, 2005.

[10] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 1–16. Springer-Verlag, 1998.

[11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[12] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[13] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 6:68–93, 1984.

[14] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In E. Emerson and K. Namjoshi, editors, *Proc. of the $7^t h$ workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lect. Notes in Comp. Sci.*, pages 364–380. Springer-Verlag, 2006.

[15] A. Pnueli and R. Rosner. A framework for the synthesis of reactive modules. In F. Vogt, editor, *Proc. Intl. Conf. on Concurrency: Concurrency 88*, volume 335 of *Lect. Notes in Comp. Sci.*, pages 4–17. Springer-Verlag, 1988.

[16] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.

[17] M. Rabin. *Automata on Infinite Objects and Churc's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972.

[18] P. Ramadge and W. Wonham. The control of discrete event systems. *Proc. of the IEEE on Control Theory*, 77:81–98, 1989.

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# Security, Privacy, Usability and Reliability (SPUR) in Mobile Networked Embedded Systems: The Case of Modern Automobiles

K. Venkatesh PRASAD [a,1] and TJ GIULI [a]
[a] *Ford Research and Advanced Engineering*

**Abstract.** The notion of nearly all things being networked almost all the time has been with us for over two decades now. Most networked entities had been fixed until the advent of cellular telephony. With cellular phones the fringes of the information age have begun to expand in pervasive forms. Adding the automobile to this context adds a new dimension to this domain of pervasive networks. In this chapter we introduce the notion of a *mobile networked embedded systems* (MNES) in which a mobile entity such as an automobile is composed of internally as well as externally networked software components. We further discuss the challenges going forward of designing a MNES-vehicle with regard to security, privacy, usability, and reliability (SPUR).

**Keywords.** Embedded software, security, privacy, mobile computing

## Introduction

Modern automobiles make a good pedagogical case for study of mobile networked embedded systems (MNES) in general, and for the study of security, privacy, reliability and privacy (SPUR), in particular. Today's automobiles may contain more than 10 million lines of code distributed across several tens of embedded processors, in inter-connected us as many as 6 distinct wired and wireless networks, making automobiles a rich networking environment (Figure 1). All this computing and communications capability is housed in highly mobile and durable platforms, and driven by a wide range of users, in all types of physical and electronic environmental conditions. There are numerous other examples of embedded systems such as toasters, automatic teller machines or aircraft, but toasters don't have much of a SPUR concern, automatic teller machines have considerable SPUR requirements but do not (usually) move and aircraft have SPUR requirements but are operated only by highly trained personnel. The automobile therefore stands out as particularly attractive for the study of SPUR in MNES.

Table 1 contains examples of several real-world automotive use-cases, along with their associated SPUR attributes. The relevance of each attribute — security (S), privacy

[1]2101 Village Rd. MD 2122, Dearborn, MI 48103, USA

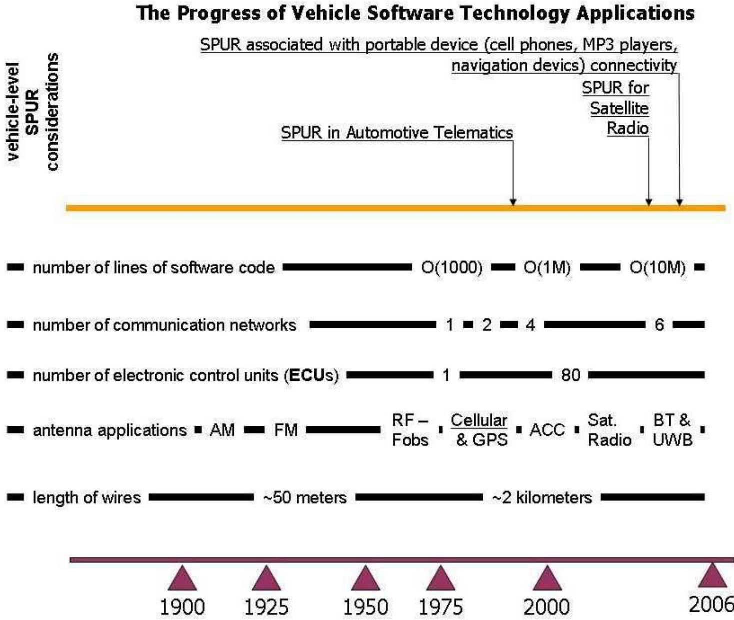



**Figure 1.** The increasing influence of software on vehicle design. In 1975, software was mostly limited to powertrain control and was a few thousand lines of code in size. By the year 2000, software had grown to several hundred thousand lines of code and by 2006 code has grown to several hundred megabytes of code. As we move into the next quarter century, we see the influence of software continuing to grow with its cost together with associated electronics at about 20-30% of vehicle cost and its benefits being derived from the fact that 90% of future features are expected to be dependant software.

**Table 1.** Examples illustrating how each SPUR attribute relates to an example from the automotive domain. Each attribute is classified as having low (l), medium (m) or high (h) relevance in relation to the example.

| Example | S | P | U | R |
|---|---|---|---|---|
| Hands-free Bluetooth | h | h | h | m |
| Remote diagnostics | h | h | l | m |
| Stolen vehicle tracking | h | h | l | h |
| Emergency blinkers | l | l | h | h |

(P), usability (U) and reliability (R) — is indicated by a low (l), medium (m) or high (h) measure. These measures were arrived at by subjective reasoning and are meant to serve as examples. The next section deals with service-oriented architectures, followed by an overview of communications systems in automobiles. Subsequent sections deal with an introduction to SPUR and conclusions.

## 1. Embedded Service-Oriented Architectures

Modern automobiles have become quite complex in terms of the number of microprocessors, lines of code, and the size of internal vehicle networks. However, while the amount of compute hardware and software has increased, the methodologies used in implementing software on most vehicles has lagged behind the state of the art typically seen in

the IT industry. Vehicle modules are typically designed as black boxes, where modules may be designed and manufactured by different suppliers. Modules communicate with each other using network messages, but these messages are usually not part of an industry standard application programming interface (API), but rather a collection of application specific agreed-upon messages that may change between products within the same original equipment manufacturer (OEM), even if the module functionality remains the same.

Furthermore, modules often contain overlapping implementation functionality. For example, a luxury vehicle might have a navigation system with a text-to-speech component that reads directions aloud. The same vehicle might also have a hands-free-phone module with a separate text-to-speech component that tells the driver who is calling. This overlapping functionality increases development costs because the function was developed twice and it may also impact the unit cost of each module because twice as much memory is necessary to store the program code.

Additionally, as consumers demand more features, the complexity of inter-module interactions is also growing. A safety system that detects a crash has occurred can deploy the airbags, unlock the doors, and instruct the telematics system to call emergency services. This functionality requires a high level of integration between modules that do not necessarily fall under the category of "safety systems," which necessitates at least rudimentary interfaces allowing modules to communicate. If these interfaces are not well-designed, a change in any one of these modules could necessitate changes in all of the other modules.

Service-oriented architectures (SOA) have been used in other domains to solve these types of problems, and may find successful application in the automotive domain as well [1]. A service-oriented architecture is one in which loosely-coupled components interact through well-specified interfaces and APIs using standard network protocols. Software components are decomposed into *services* that export functionality to other components as well as consume the services of other components through well-defined interfaces. Ideally, services are single-purpose software components that can easily be composed with other services to create larger programs. The benefits of this type of design include:

- *Reduced dependencies between module implementations:* A key benefit of having well-designed interfaces between modules is that underlying module implementations may change while leaving the interfaces unchanged. Thus, changes to a module's implementation do not necessitate code changes to other modules it interacts with, resulting in significant cost savings.
- *Software reuse:* To build a complex feature like a navigation system in a service-oriented architecture, a software engineer composes a set of services together along with core logic that specifies the behavior of the feature. A navigation system could call on a GPS service that keeps track of the vehicle's current location, a text-to-speech service that reads text to the driver, a route-planning service, and a map service that displays map graphics. Although the text-to-speech service is used by the navigation feature, the aforementioned hands-free phone feature could also use the text-to-speech service as well, thus eliminating the need for both the navigation module and hands-free phone module to implement their own text-to-speech code. Service-oriented architectures promote software reuse by advocating general services that are usable by multiple software programs.

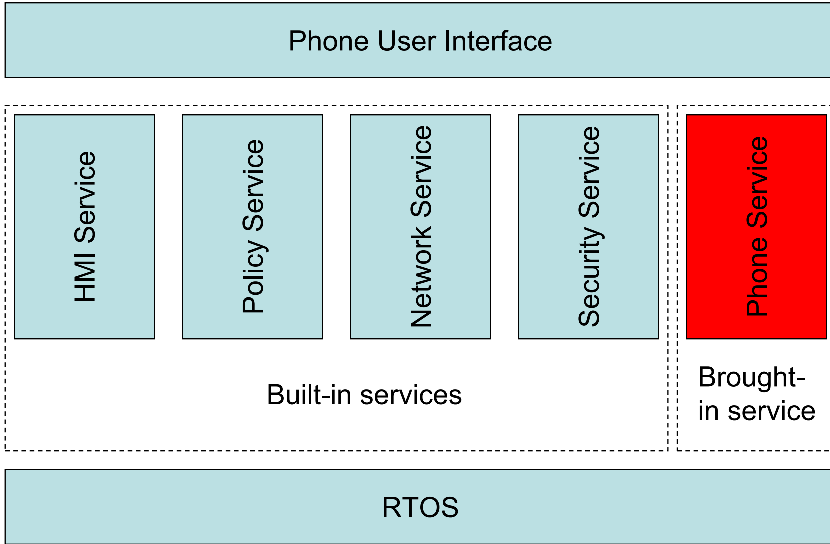



**Figure 2.** A phone application built on top of a service-oriented architecture. The built-in components include HMI, policy, network, and security services. The HMI service controls user-interface controls such as the speakers and display.

- *Consistency of user interface:* There is an added benefit to reusing user-visible services. By using the same text-to-speech engine for both the navigation and hands-free-phone feature, the driver hears the same voice consistently across features. This enhances the feel of feature integration and makes the voice system more usable.

### 1.1. Example

With the increasing deployment of high-bandwidth wireless network technologies such as EVDO, WiMAX, and WiFi hotspots, the possibility of vehicles utilizing broadband connectivity to download software, firmware updates, and digital content is fast approaching. In this example, we explore the implications of such a connection by examining a software download use case.

In this use case, a driver brings a new cell phone into his vehicle and wants to integrate it into his driving experience. He downloads software from the OEM, enabling the vehicle to make and receive hands-free calls and notify the driver of any appointments on his phone's calendar. The software displays a cell phone interface on an in-vehicle display, allowing the driver to control his cell phone either through the built-in knobs, buttons, and touchscreen display, or via voice commands.

Figure 2 shows an architecture diagram illustrating how a service-oriented architecture might look in a vehicle. The HMI, policy, network, and security services are all core built-in services. The HMI service displays data via any built-in displays and it gathers input from controls such as knobs and buttons. The policy service regulates how user interfaces may interact with the driver given the current context. For example, the policy service would prevent a DVD from being played on a driver-visible LCD screen unless the car is parked. The network service provides network connectivity to other ser-

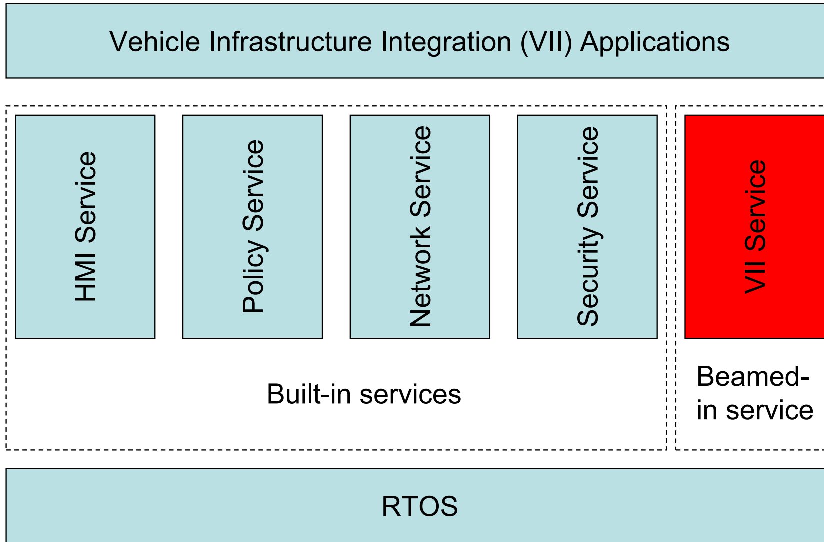



**Figure 3.** Wireless ("beamed-in") vehicle to (roadside) infrastructure communications introduces added complexity to vehicle software as security, privacy, usability and reliability need to be reviewed in this context.

vices and applications. The security service provides services such as authentication and encryption. The phone service is a brought-in service, available when a cell phone is present. The phone user interface uses the phone service to make and receive calls and it uses the HMI service to display a user interface on an in-vehicle display as well as broadcast sound from calls over the vehicle's speakers.

In addition to offering the ability to send and receive phone calls, the phone service may also offer data transport (e.g., over a 3G network), so the network service could use the phone service to provide another means of connectivity. Thus if the vehicle is outside of metro-area WiMAX connectivity, the vehicle can still have network connectivity as long as the cell phone is powered on.

## 2. Overview of Communications Systems in Automobiles

As shown in Figure 1 automobiles today may have up to 6 communications networks. A review of intra-vehicle communications may be found in [2] and a representative selection of inter-vehicle and vehicle-to-infrastructure communications may be found in [3].

Broadly stated, one could group automobile communications into the following categories:

- Intra-vehicle communications: Vehicle modules communicate mostly over wired networks following standard protocols such as the controller area network or CAN, local interconnect network or LIN and media-oriented systems transport or MOST. These networks support a wide range of applications, each with a distinct set of functional and non-functional requirements spanning major vehicle domains such as powertrains, chassis, body, safety, security, and information and

entertainment domains. Such networks have also traditionally supported interconnections to (wireless) radio-frequency units such as security fobs and more recently to tire pressure monitoring systems.

- Vehicle-to-infrastructure communications: A popular example of this aspect of vehicle communications is automobile telematics, where a vehicle modules communicates to a remote call and data center via a cellular wireless communications protocol. This is rapidly being augmented by various vehicle to roadside infrastructure communications protocols such as the dedicated short-range communications or DSRC (a 5.9GHz) protocol. etc.

### *2.1. Example*

The Vehicle Infrastructure Integration (VII) project is a collaboration between the United States Department of Transportation, ten State Departments of Transportation and the top eight automobile manufacturers that sell vehicles in the United States. The aim of VII is to enable vehicle to vehicle and vehicle to infrastructure communications to support safety applications and traffic monitoring [4].

Several use cases have been developed for VII including:

- *Emergency electronic brake lights* Drivers are frequently only able to see the taillights of the vehicle directly in front of them. If someone two or three vehicles ahead brakes hard, drivers several cars behind will not know that they need to brake until the car directly in front of them brakes. In this use case, whenever a vehicle brakes, it sends out a vehicle-to-vehicle broadcast to the vehicles behind it that it is slowing down, giving drivers an early warning.
- *Highway signage* In this use case, transmitters along roadways broadcast highway signage information such as exit numbers and nearby businesses. This information is displayed for the driver on the vehicle's HMI.
- *Electronic payment* In this scenario, vehicles communicate with infrastructure such as roadway transmitters and gas stations to pay tolls and purchase fuel.

VII services could be implemented in a service-oriented manner, as seen in Figure 3.

## 3. Implications of Security, Privacy, Usability, and Reliability (SPUR)

Figure 1 illustrates the increasing importance of software on vehicle design, both in terms of development costs and in impact on the driver experience. Software systems enable advanced traction control, more efficient engines, advanced onboard diagnostics, and infotainment features such as navigation. Reliance on increasingly complex software to provide these advanced features means that automobile manufacturers must also be concerned about some of the same issues that concern software producers, among them security, privacy, usability, and reliability. To be sure, these are all concerns that automotive manufacturers have been aware of in the context of physical systems (e.g., locks and immobilizers to prevent vehicle theft, well-designed interior controls to enhance usability, and high quality components to ensure vehicle reliability). We can discuss the problem of automotive embedded security in terms of two dimensions: network security and software security. As we will see, both network security and software security are necessary to ensure correct vehicle operations.

Section 2 discussed the need for both vehicle-to-vehicle as well as vehicle-to-infrastructure communications. In the case of active safety applications such as crash mitigation and avoidance, the authenticity and accuracy of vehicle communications is crucial. For example, in the VII early brake warning scenario, a vehicle broadcasts a notification whenever its driver brakes. Drivers that cannot see the braking vehicle's taillights are notified that a vehicle in the lane ahead of them is slowing down and so they should be prepared to slow down as well. If a malicious actor can arbitrarily spoof safety messages such as the early brake warning, they could potentially snarl traffic by injecting false messages into the system.

One potential solution to this problem is to require all vehicles to sign their safety-related messages with digital signatures. These signatures would prove that they had originated from a specific vehicle and would be generated using certificates issued by vehicle manufacturers or possibly local departments of transportation as part of vehicle registration. Requiring valid signatures on all vehicle safety messages would make it much harder for a malicious adversary to spoof these types of messages.

Other forms of authentication are necessary in vehicle communications. For example, some telematics systems in production today allow service centers to remotely unlock specific vehicles if a driver has locked their key in their vehicle. To prevent car thieves from stealing a car using a false door-unlock signal, an authentication mechanism should allow the vehicle to authenticate that a door-unlock message is actually coming from a valid service center.

In addition to communication security, vehicle software should be hardened against intrusion to prevent subversion by a malicious adversary. In the case of the early brake scenario, if a virus can compromise the vehicle's safety software, it could send out signed brake notifications even when the driver is not braking. To ensure the early brake system works well, therefore, vehicles' software systems must be resistant to intrusion and communication between vehicles must be difficult to falsify.

Driver privacy is also an issue in designing vehicle-to-vehicle and vehicle-to-infrastructure communications systems. If communications from vehicles include location information such as the current GPS coordinates of a vehicle and a uniquely identifying characteristic such as a license plate number, it is possible to track drivers. Many people find this type of location surveillance unacceptable and would resist buying vehicles that allowed them to be tracked. Potential solutions to this problem involve anonymizing individual vehicle communications through other vehicles acting as proxies [5].

Usability is also important when considered in the context of security and privacy. Different drivers can have different views on what they consider private, therefore they should be able to specify how their data should be protected. An online web portal could be one method for a driver to specify his preferences, or there could also be an in-vehicle component as well. Either way, the driver should be able to easily communicate their preferences to the vehicle. Additionally, the vehicle needs to clearly communicate security and privacy related events to the driver. For example, if the vehicle detects the presence of a software virus, it should communicate this to the driver in such a way as to not alarm the driver while the vehicle is moving.

Reliability has traditionally been a key concern of automobile manufacturers in the context of mechanical and electrical systems, and that emphasis now translates into software systems. Because software now interacts with most major vehicle subsystems, a software bug could potentially affect the drivability of a vehicle. Furthermore, even if a

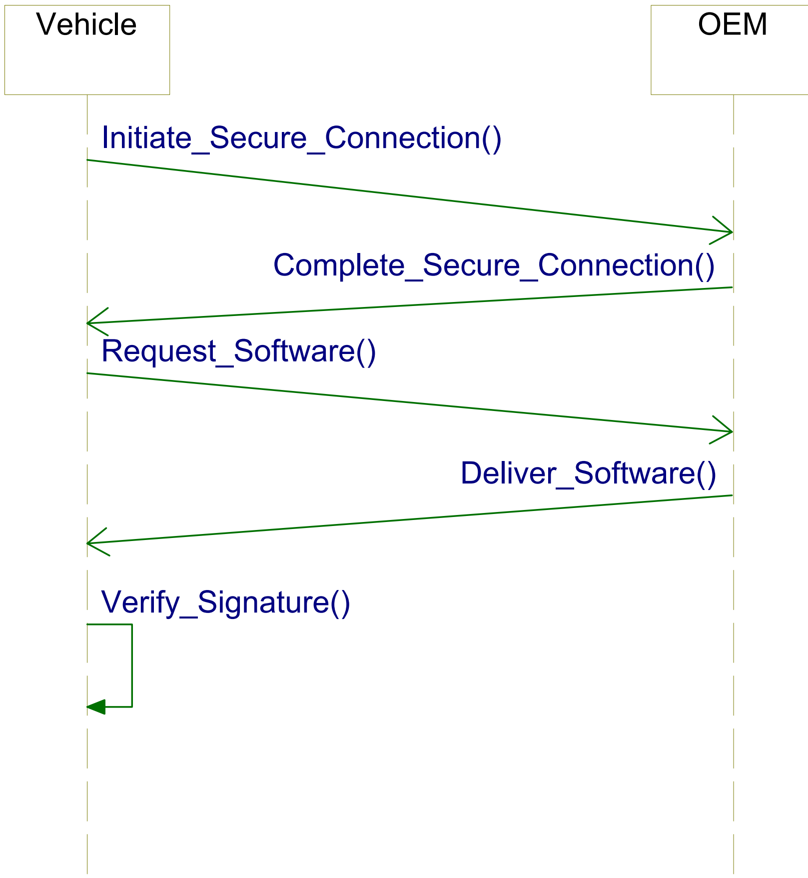



**Figure 4.** A sequence diagram showing how a vehicle acquires a new piece of software from an original equipment manufacturer (OEM).

software failure has nothing to do with the powertrain, such as a bug in a rear-seat DVD player, consumers may incorrectly assume that powertrain software is also buggy.

Each of the individual SPUR attributes are important on their own, but because automobiles are complex systems, the SPUR attributes must be analyzed as a whole, to ensure that strengthening one attribute does not weaken another. For example, a module that maintains a driver's phonebook from their cell phone could prompt the driver for a password every time it downloads the phone book. However, this could negatively affect the usability of the system and frustrate drivers. For a more in-depth look at how SPUR affects future automobile design, see [6,7].

### 3.1. *Example*

We will now consider the example from Section 1.1 with a focus on SPUR. Recall that our example has to do with a driver downloading software to better interface between his vehicle and his cell phone. Figure 4 shows the sequence of interactions between the vehicle and its OEM. To download the software, the vehicle first initiates a secure connection

to the OEM using a wireless network connection. The vehicle and the OEM mutually authenticate each other so that the vehicle can confirm that it is actually talking to its OEM and the OEM knows that the vehicle is one of its own. After authenticating each other, the vehicle and OEM set up an encrypted connection. The vehicle then requests the software package and the OEM delivers the software. The software is digitally signed by the OEM and the vehicle verifies the signature before installation. Finally, the software is ready for use.

There are clearly several aspects of the software download and installation process that have to do with security. First, the authentication step is necessary to prevent a malicious party from posing as the OEM and offering malign software. Encrypting the connection prevents eavesdroppers in case the vehicle transmits sensitive information to the OEM. As an extra security measure, the software is signed by the OEM and verified by the vehicle, again to ensure that the software comes from a trusted source.

Both the authentication and signature verification are intended to prevent malicious software from being downloaded to the vehicle that could negatively affect the driving experience. In our example, if the cell phone integration software is in fact malicious spyware, the software could download the driver's contact list and text messages off of his phone and email the information to cybercriminals. Or the software could potentially interfere with the operation of the vehicle directly.

The issue of user data brings up the issue of privacy. In some instances, the driver might be perfectly happy exposing the contact list in his phone, for instance, to a friend. However, giving the contact list to a stranger would be unacceptable. As consumer devices like cell phones become more integrated with automobiles, auto manufacturers must take steps to ensure that data considered private by customers remains private.

The primary benefit to the driver for downloading the cell phone integration software is enhanced usability. He can now make and receive phone calls while keeping both hands on the wheel and his eyes on the road by speaking commands to the phone. However, there are also other usability concerns in our example. For instance, how usable is the software download process? Does the vehicle automatically detect the make and model of the driver's cell phone and download the correct software, or must the driver be involved? What happens if there is a security exception, e.g., the software's signature is invalid? How does the user specify privacy preferences? These issues must all be resolved in a usable manner.

Software reliability is a concern in our example as well. The security mechanisms prevent overtly malicious from being executed by the vehicle, however unintentionally buggy software can cause just as much damage [8]. For example, a bug in the cell phone integration software could accidentally delete the driver's cell phone contact list and text messages. The threat of unreliable software is difficult to protect against because it requires extensive resources to ensure software quality, such as formal verification [9] and software quality assurance. The solution to this problem in the PC software industry has so far been to provide frequent software upgrades to fix bugs. This method is more challenging in the automotive domain because vehicles may not always have network connections, and even if they do, the cost to the consumer may be much greater than they are used to with their PC.

## 4. Conclusion

This chapter has introduced the need to assess security, privacy, usability and reliability or SPUR in the context of mobile networked embedded systems, using automobiles as an example. Embedded service-oriented architectures and communications systems within automobiles were reviewed in the context of SPUR along with several examples.

## Acknowledgements

The authors would like to thank Frank Perry and David Watson for their helpful insights into SPUR and VII.

## References

[1] Krüger, I.H., Gupta, D., Mathew, R., Moorthy, P., Phillips, W., Rittmann, S., Ahluwalia, J.: Towards a process and tool-chain for service-oriented automotive software engineering. In: Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems. (2004)
[2] Navet, N., Song, Y., Simonot-Lion, F., Wilwert, W.: Trends in automotive communication systems. Proceedings of the IEEE **93**(6) (2005) 1204–1223
[3] IEEE Wireless Communications **13**(5) (2006) c1–c1
[4] Farkas, K.I., Heidemann, J., Iftode, L., Kosch, T., Strassberger, M., Laberteaux, K., Caminiti, L., Caveney, D., Hada, H.: Vehicular communication. IEEE Pervasive Computing **5**(4) (2006) 55–62
[5] Sampigethaya, K., Huang, L., Li, M., Poovendran, R., Matsuura, K., Sezaki, K.: CARAVAN: Providing location privacy for VANET. In: Embedded Security in Cars (ESCAR). (2005)
[6] Prasad, K.V., Giuli, T., Watson, D.: The case for modeling security, privacy, usability and reliability (SPUR) in automotive software. In: Automotive Software Workshop, San Diego, CA (2006)
[7] Giuli, T., Watson, D., Prasad, K.V.: The last inch at 70 miles per hour. IEEE Pervasive Computing **5**(4) (2006) 20–27
[8] Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. Computer **26**(7) (1993) 18–41
[9] Meadows, C.: The NRL protocol analyzer: An overview. Journal of Logic Programming **26**(2) (1996) 113–131

*Software System Reliability and Security*
*M. Broy et al. (Eds.)*
*IOS Press, 2007*
*© 2007 IOS Press. All rights reserved.*


# A Verifying Compiler for a Multi-threaded Object-Oriented Language

K. Rustan M. Leino and Wolfram Schulte
*Microsoft Research,Redmond, WA, USA*
*e-mail: {leino,schulte}@microsoft.com*

**Abstract.** A verifying compiler automatically verifies the correctness of a source program before compiling it. Founded on the definition of the source language and a set of rules (a methodology) for using the language, the program's correctness criteria and correctness argument are provided in the program text by interface specifications and invariants.

This paper describes the program-verifier component of a verifying compiler for a core multi-threaded object-oriented language. The verifier takes as input a program written in the source language and generates, via a translation into an intermediate verification language, a set of verification conditions. The verification conditions are first-order logical formulas whose validity implies the correctness of the program. The formulas can be analyzed automatically by a satisfiability-modulo-theory (SMT) solver.

The paper defines the source language and intermediate language, the translation from the former into the latter, and the generation of verification conditions from the latter. The paper also builds a methodology for writing and verifying single- and multi-threaded code with object invariants, and encodes the methodology into the intermediate-language program.

The paper is intended as a student's guide to understanding automatic program verification. It includes enough detailed information that students can build their own basic program verifier.

## 0. Introduction

A *verifying compiler* is a compiler that establishes that a program is correct before allowing it to be run. Verifying compilers can come in many flavors, from systems that generate provably correct code from specifications to systems that ask users to guide an interactive theorem prover to produce a replay-able proof script. In this paper, we consider a verifying compiler that automatically generates logical proof obligations, *verification conditions* (VCs), from a given program, its embedded specifications, and a set of rules (a *methodology*) that guides the use of the language. The validity of the VCs implies the correctness of the program. The VCs are passed to a satisfiability-modulo-theory (SMT) solver to be discharged automatically, if possible. Failed proof attempts are presented to users as error messages, to which a user responds by fixing errors or omissions in the program and its specifications.

The Spec$^\sharp$ programming system [6] is a modern research prototype of such a verifying compiler. It consists of an object-oriented programming language (also called Spec$^\sharp$) designed as a superset of the .NET programming language C$^\sharp$, enriching the type system (for example with non-null types) and adding specifications (like pre- and postconditions) as a part of the language, a methodology for using the language, a compiler that produces executable code for the .NET virtual machine, an integration into the Microsoft Visual Studio integrated development environment, and a static program verifier.

Generating verification conditions for high-level source programs is nontrivial and involves a large number of details and design decisions. Therefore, the Spec$^\sharp$ static program verifier (which is known as Boogie [4]) splits the task into two: it first translates the Spec$^\sharp$ program into an intermediate verification language (called BoogiePL [11]) and then generates VCs from it. This lets the tool designer make modeling decisions in terms of the intermediate language, which thus provides a level of abstraction above the actual formulas passed to the SMT solver.

In this paper, we want to convey the design of the program-verifier component of a verifying compiler. Doing so for Spec$^\sharp$ and BoogiePL is too large of a task for the paper, so we instead define a core object-oriented source language (which we shall call Spec$^\flat$) and an imperative intermediate verification language (which we shall call BoogiePL$^\flat$). As their names suggest, these languages are representative of Spec$^\sharp$ and BoogiePL, respectively. The Spec$^\flat$ language features classes and single-inheritance subclasses, object references, dynamic method dispatch, co-variant arrays, multi-threading, and mutual-exclusion locks.

*Outline* We start from the bottom up. In Section 1, we define BoogiePL$^\flat$ and its VC generation. We define Spec$^\flat$ in Section 2, where Section 2.3 defines a translation from Spec$^\flat$ into BoogiePL$^\flat$. We then take on some hard questions of how to write specifications in such a way that one can reason about programs modularly—to scale program verification, it must be possible to specify and verify each part (say, each class) of a program separately, in such a way that the separate verification of each part implies the correctness of the entire program. In Section 3, we introduce a methodology for *object invariants*, which specify the data consistency conditions of class instances, and define a translation of the new features and rules into BoogiePL$^\flat$. In Section 4, we also add features for writing multi-threaded code, a methodology for those features, and a corresponding translation into BoogiePL$^\flat$. Throughout, we give enough details to build a basic program verifier for Spec$^\flat$. Concepts and typical design issues carry over to other languages as well.

*Foundational Work* Program verification has a long history. The foundation for today's verification research was laid down by Floyd's inductive assertion method [25], Hoare's axiomatic basis for programming [30], and Dijkstra's characterization of semantics [18]. Early program verifiers include the systems of King [40,39], Deutsch [15], Good *et al.* [28], and German [27], the Stanford Pascal Verifier [56], and the Ford Pascal-F Verifier [63]. Two verifying compilers for procedural languages are SPARK [2] and B [0].

$$
\begin{array}{l}
\mathbf{const}\ K\colon \mathbf{int}\ ;\\
\mathbf{function}\ f(\mathbf{int})\ \mathbf{returns}\ (\mathbf{int})\ ;\\
\mathbf{axiom}\ (\exists\, k\colon \mathbf{int} \bullet\ f(k) = K\ )\ ;\\
\mathbf{procedure}\ \mathit{Find}(a\colon \mathbf{int},\ b\colon \mathbf{int})\ \mathbf{returns}\ (k\colon \mathbf{int})\ ;\\
\quad \mathbf{requires}\ a \leqslant b \land (\forall\, j\colon \mathbf{int} \bullet\ a < j \land j < b \Rightarrow f(j) \neq K\ )\ ;\\
\quad \mathbf{ensures}\ f(k) = K\ ;\\
\mathbf{implementation}\ \mathit{Find}(a\colon \mathbf{int},\ b\colon \mathbf{int})\\
\{\ \mathbf{assume}\ f(a) = K\ ;\ k := a\\
[]\ \mathbf{assume}\ f(b) = K\ ;\ k := b\\
[]\ \mathbf{assume}\ f(a) \neq K \land f(b) \neq K\ ;\ \mathbf{call}\ k := \mathit{Find}(a-1, b+1)\\
\}
\end{array}
$$

**Figure 0.** An example BoogiePL$^\flat$ program, showing the declaration of a constant $K$, a function $f$, an axiom that says $f$ has a $K$ element, a procedure $\mathit{Find}$ that finds the $K$ element of $f$, and a recursive implementation of $\mathit{Find}$. The call statement **call** $x := \mathit{Find}(0,0)$ will set $x$ to some $K$ element of $f$.

## 1. An Intermediate Imperative Verification Language

This section defines BoogiePL$^\flat$, an intermediate language for program verification. BoogiePL$^\flat$ is essentially BoogiePL [11], but without some of the more advanced features of BoogiePL. A BoogiePL$^\flat$ program consists of two parts:

- a mathematical part to define a logical basis for the terms used in the program, described by constants, functions, and axioms, and
- an imperative part to define the behavior of the program, described by procedure specifications, mutable variables, and code.

Figure 0 shows a simple BoogiePL$^\flat$ program. The mathematical part of this program are the declarations of $K$, $f$, and the axiom. The imperative part of the program is given by the specification and implementation of $\mathit{Find}$.

The semantics of a BoogiePL$^\flat$ program is defined as a logical formula, consisting of the theory induced by the mathematical part and the semantics induced by each procedure implementation in the imperative part. The program is considered correct if the logical formula is valid.

The next subsections introduce BoogiePL$^\flat$: its type system, the syntax of its mathematical and imperative parts, and the semantics of the code.

### *1.0. Basic Concepts*

*Backus Naur Form* We use the common Backus Naur form to specify syntax. Nonterminals are written in italics. Terminals are keywords (written in bold), symbols (written as themselves), and a set *Id* of identifiers. For any nonterminal $a$, the suffixes $a^?$ denotes either the empty word or $a$, $a^+$ denotes one or more repetitions of $a$, and $a^*$ denotes either the empty word or $a^+$. Depending on the context, repetitions are separated by commas (*e.g.*, in argument lists) or by white space (*e.g.*, in a sequence of declarations); this is not further specified.

*Program Structure* At the top level, a BoogiePL$^\flat$ program is a set of declarations.

$$\begin{array}{rcl} \mathit{Program} & ::= & \mathit{Decl}^* \\ \mathit{Decl} & ::= & \mathit{Constant} \mid \mathit{Function} \mid \mathit{Axiom} \\ & \mid & \mathit{Variable} \mid \mathit{Procedure} \mid \mathit{Implementation} \end{array}$$

*Type System* Value-holding entities in BoogiePL$^\flat$ are typed, despite the fact that a theorem prover used on BoogiePL$^\flat$ programs may be untyped. The purpose of semantic-less types in BoogiePL$^\flat$, like the purpose of explicit declarations of variables and functions, is to guard against certain easy-to-make mistakes in the input.

There are four built-in basic types, map types, and the supertype **any**:

$$\mathit{Type} \quad ::= \quad \mathbf{bool} \mid \mathbf{int} \mid \mathbf{ref} \mid \mathbf{name} \mid \ [\ \mathit{Type}^+\ ]\ \mathit{Type} \mid \mathbf{any}$$

The type **bool** represents the boolean values **false** and **true**. The type **int** represents the mathematical integers. The type **ref** represents object references. One of its values is the built-in literal **null**. The only operations defined by the language on **ref** are equality and dis-equality tests. The type **name** represents various kinds of defined names (like types and field names). The only operations defined by the language on **name** are equality and dis-equality tests and the partial order $<:$. In a map type, the domain (that is, the types of the arguments) is given first, followed by the range type.

Type **any** represents the un-tagged union of the other types. Every type can implicitly be converted to and from the type **any**. Because types in BoogiePL$^\flat$ are semantic-less, we use the identity for these conversions. But note that the implicit conversion from **any** to a type $T$ is "unsafe" (since **any** is not a tagged union with checked tags). It is our responsibility to guarantee correct usage of expressions of type **any**.

We say a type $T$ is *assignable* to a type $U$ if $T$ is $U$ or if either $T$ or $U$ is **any**.

*Scope Rules* BoogiePL$^\flat$ supports nested lexicographic scoping, which means that (0) all identifiers introduced by top-level declarations must be distinct, and (1) identifiers introduced in inner scopes hide identifiers in outer scopes. During name resolution, an identifier is first looked up in the innermost scope, then the enclosing scope, and so on. It is an error if an identifier can't be found in the scope of its use.

### 1.1. Theories

The mathematical part of the language (constants, functions, axioms) is similar to other specification languages, including Larch [29] or the input language of the theorem prover Simplify [12].

*Constants* and *functions* are identifiers that, throughout the interpretation of a program, have a fixed, but possibly unknown, meaning.

$$\begin{array}{rcl} \mathit{Constant} & ::= & \mathbf{const}\ \mathit{Id}\ :\ \mathit{Type}\ ; \\ \mathit{Function} & ::= & \mathbf{function}\ \mathit{Id}\ (\ \mathit{Type}^*\ )\ \mathbf{returns}\ (\ \mathit{Type}\ ); \end{array}$$

Both can be used in expressions and commands.

To constrain the values of constants and functions, one uses *axioms*:

$$\mathit{Axiom} \quad ::= \quad \mathbf{axiom}\ \mathit{Expr}\ ;$$

The given expression must be of type **bool** and must not have any free variables. An axiom that comes from free is that all constants of type **name** have distinct values.

### 1.2. Variables and Procedures

The *state space* of a BoogiePL$^\flat$ program is defined by variables. A *global variable* is a variable that is accessible to all procedures.

$$Variable \quad ::= \quad \textbf{var}\ Id\ :\ Type\ ;$$

A *procedure* is a name for a parameterized operation on the state space.

$$\begin{array}{rcl} Procedure & ::= & \textbf{procedure}\ Id\ Signature\ ;\ Specification^* \\ Signature & ::= & (IdType^*)\ \textbf{returns}\ (IdType^*) \\ IdType & ::= & Id\ :\ Type \\ Specification & ::= & \textbf{requires}\ Expr\ ; \\ & | & \textbf{modifies}\ Id^*\ ; \\ & | & \textbf{ensures}\ Expr\ ; \end{array}$$

The signature defines the list of in-parameters and then the list of out-parameters.

The procedure specification consists of a number of **requires**, **modifies**, and **ensures** clauses. The expressions given by the **requires** and **ensures** clauses must be of type **bool**. Every *Id* mentioned in a **modifies** clause must name a global variable. The in-parameters are in scope in the **requires** clause, and both in- and out-parameters are in scope in the **ensures** clause.

Each **requires** clause specifies a *precondition*, which must hold at each call to the procedure (we shall see calls later). An implementation of the procedure is allowed to assign to a global variable only if it is listed in a **modifies** clause of the procedure's specification. Each **ensures** clause specifies a *postcondition*, which must hold on exit from any implementation of the procedure. The expression in an **ensures** clause is a *two-state* predicate, which means that it can refer to both the initial and final states of the procedure (using **old** expressions for the initial state, as we shall see later). The **ensures** condition thus specifies a relation between the initial and final states of the procedure.

Procedures can be given implementations.

$$Implementation \quad ::= \quad \textbf{implementation}\ Id\ Signature\ Block$$

Here, *Id* must refer to a declared procedure and *Signature* must be identical to that of the declared procedure. There are no restrictions on the number of implementations that one procedure can have; each implementation is verified to obey the same specification.

Variables come in five flavors: global variables, in-parameters, out-parameters, local variables, and quantifier-bound variables. We say that a variable is *writable* in an implementation if it is a local variable, out-parameter, or a global variable mentioned in the modifies clause.

### 1.3. Motivation for Choice of Commands

BoogiePL$^\flat$ commands have been designed to be simple and primitive. The design makes heavy use of three useful, but perhaps less known, commands: **assert**, **assume**, and **havoc**. Before we define these and other commands in the next subsection, we give a couple of examples to develop an intuitive understanding of these commands.

Let us look at how we translate Spec$^\flat$'s conditional and while loop into BoogiePL$^\flat$. Spec$^\flat$ has the usual conditional statement, written as **if** $(E)$ $S$ **else** $T$. It goes wrong if $E$ is not defined; otherwise, if $E$ evaluates to **true**, then the conditional statement executes $S$, else the conditional statement executes $T$. The translation of the conditional statement into BoogiePL$^\flat$ is defined as follows:

$$
\begin{array}{l}
\mathit{Tr}[\![\textbf{if } (E)\ S \textbf{ else } T]\!] = \\
\quad \textbf{assert } \mathit{Df}[\![E]\!]\ ; \\
\quad \{\ \textbf{assume } \mathit{Tr}[\![E]\!]\ ;\ \mathit{Tr}[\![S]\!] \\
\quad [\!]\ \textbf{assume } \neg \mathit{Tr}[\![E]\!]\ ;\ \mathit{Tr}[\![T]\!] \\
\quad \}
\end{array}
$$

The translation uses three functions (cf. Section 2 for their full definitions). The function $\mathit{Tr}[\![s]\!] = c$ translates a Spec$^\flat$ statement $s$ into the BoogiePL$^\flat$ command $c$. The functions $\mathit{Df}[\![e]\!] = e'$ and $\mathit{Tr}[\![e]\!] = e''$ return two BoogiePL$^\flat$ expressions for one Spec$^\flat$ expression: $e'$ says whether $e$ is defined and, if so, $e''$ denotes its translated value.

The translation uses an **assert** command to check that $E$ is defined. If $E$ is not defined, that is, if $\mathit{Df}[\![E]\!]$ evaluates to **false**, then the **assert** command will cause the program to halt with an error.

The rest of the translation consists of a nondeterministic choice, as denoted by $[\!]$. Each choice begins with an **assume** command, which indicates under which condition the remainder of that path of the program is analyzed. That is, the translation of $S$ is analyzed only if $\mathit{Tr}[\![E]\!]$ evaluates to **true**, and analogously for $T$.

Spec$^\flat$'s while loop **while** $(E)$ **invariant** $J$; $\{S\}$ proceeds as follows. The while loop goes wrong if the loop invariant $J$ is not defined or evaluates to **false**, and it goes wrong if the loop condition $E$ is not defined. Otherwise, if $E$ evaluates to **true**, the body $\{S\}$ is executed, after which the entire while loop is executed again. If $E$ evaluates to **false**, the while loop terminates. We translate the while loop into BoogiePL$^\flat$ as follows:

$$
\begin{array}{l}
\mathit{Tr}[\![\textbf{while } (E) \textbf{ invariant } J;\ \{S\}]\!] = \\
\quad \textbf{assert } \mathit{Df}[\![J]\!]\ ; \textbf{assert } \mathit{Tr}[\![J]\!]\ ; \\
\quad \textbf{havoc } \mathit{Md}[\![S]\!]\ ; \\
\quad \textbf{assume } \mathit{Df}[\![J]\!] \wedge \mathit{Tr}[\![J]\!]\ ; \\
\quad \textbf{assert } \mathit{Df}[\![E]\!]\ ; \\
\quad \{\ \textbf{assume } \mathit{Tr}[\![E]\!] \\
\quad\quad \mathit{Tr}[\![\{S\}]\!]\ ; \\
\quad\quad \textbf{assert } \mathit{Df}[\![J]\!]\ ; \textbf{assert } \mathit{Tr}[\![J]\!]\ ; \\
\quad\quad \textbf{assume false} \\
\quad [\!]\ \textbf{assume } \neg \mathit{Tr}[\![E]\!] \\
\quad \}
\end{array}
$$

The translation uses a function $\mathit{Md}[\![S]\!]$, which returns the list of variables possibly modified by $S$, also known as the *syntactic targets* of the loop.

The translation can be understood as follows. First, the loop invariant is checked on entry to the loop. Then, we want to look at just one iteration of the loop, but we want it to be an arbitrary iteration. The translation thus "fast forwards" to an arbitrary iteration by setting the variables to arbitrary values. More precisely, the translation sets the syntactic targets of the loop to arbitrary values (**havoc** $\mathit{Md}[\![S]\!]$) satisfying the loop invariant (**assume** $\mathit{Df}[\![J]\!] \wedge \mathit{Tr}[\![J]\!]$). In that arbitrary iteration, the translation checks that

the loop condition is defined, and then either performs one more iteration or terminates the loop. After executing $S$, the translation checks that the invariant $J$ still holds—this is essentially the inductive step of the loop verification.

One thing remains to be explained: The **assume false** command at the end of the first choice branch indicates that the remainder of the program is not analyzed immediately after an arbitrary loop iteration. The analysis proceeds under the assumption of successful termination, which happens through the second choice branch.

### 1.4. Commands

Now we are ready to introduce the BoogiePL$^\flat$ commands, which follow this grammar:

$$
\begin{array}{rcl}
\mathit{Command} & ::= & \mathbf{assert}\ \mathit{Expr} \\
& | & \mathbf{assume}\ \mathit{Expr} \\
& | & \mathbf{havoc}\ \mathit{Id}^{+} \\
& | & \mathit{Designator}\ :=\ \mathit{Expr} \\
& | & \mathbf{call}\ \mathit{Id}^{*}\ :=\ \mathit{Id}\ (\ \mathit{Expr}^{*}\ ) \\
& | & \mathit{Command}\ ;\ \mathit{Command} \\
& | & \mathit{Command}\ []\ \mathit{Command} \\
& | & \mathit{Block} \\
\mathit{Block} & ::= & \{\mathit{Variable}^{*}\ \mathit{Command}\} \\
\mathit{Designator} & ::= & \mathit{Id} \\
& | & \mathit{Designator}\ [\ \mathit{Expr}^{+}\ ]
\end{array}
$$

The command **assert** $E$ evaluates $E$, which must be of type **bool**. If $E$ evaluates to **true**, then the command terminates. If $E$ evaluates to **false**, then the command *goes wrong*, which indicates a non-recoverable error.

The command **assume** $E$ evaluates $E$, which must be of type **bool**. If $E$ evaluates to **true**, then the command terminates. If $E$ evaluates to **false**, then the execution of the program stalls forever, which entails that this program path no longer has any chance of going wrong.

In the command **havoc** $xx$, each identifier in the list $xx$ must refer to a writable variable. The command assigns an arbitrary value to every variable in $xx$.

The assignment command uses a *designator*. In general, a designator expression has one of two forms. If it is an identifier $x$, then $x$ must refer to a variable or constant. The type of such an expression is the type of $x$. A designator expression of the form $A[EE]$ requires the type of $A$ to be a map type. The number of expressions in the list $EE$ must equal the number of argument types of $A$, and the types of the expressions in $EE$ must be assignable to the types of the corresponding argument types of $A$. The type of the expression $A[EE]$ is the range type of $A$.

The designator expression used as the left-hand side $d$ of an assignment command $d := E$ must be either a writable variable or an expression $a[EE]$ where $a$ is a writable map variable. The type of $E$ must be assignable to the type of $d$. If the left-hand side is a variable, the assignment command changes the value of that variable to $E$. If the left-hand side is an expression $a[EE]$, the assignment command overwrites $a$ with a new map that is like the old except that it maps $EE$ to $E$.

In the call command **call** $w := P(EE)$, $P$ must refer to a procedure and $w$ must refer to distinct writable variables that are not mentioned in $P$'s **modifies** clauses. The

length of the list $w$ must equal the number of out-parameters of $P$, and the types of the out-parameters of $P$ must be assignable to the types of the corresponding variables in $w$. The length of the list $EE$ of expressions must equal the number of in-parameters of $P$, and the types of the expressions in $EE$ must be the assignable to the types of the corresponding in-parameters of $P$.

The call command evaluates the expressions in $EE$ and binds the resulting values to the in-parameters of $P$. It also binds $w$ to the out-parameters of $P$. The call command goes wrong if any of $P$'s declared preconditions is not satisfied. Otherwise, the call command sets $w$ and the variables in $P$'s **modifies** clauses to arbitrary values satisfying $P$'s postconditions. Note that the meaning of the call command is given by the procedure's specification alone; the procedure's implementations are separately checked against the specification, thus enabling modular reasoning.

The sequential composition of two commands $S$ and $T$ is written $S\ ;\ T$, and its behaviors are defined by the behaviors of $S$ followed by the behaviors of $T$. The choice composition of two commands is written $S\ []\ T$, and its behaviors are defined as the union of the behaviors of $S$ and $T$. That is, $S\ []\ T$ can behave as either $S$ or $T$. We let ; bind stronger than [].

The block command $\{VV\ \ S\}$ introduces local variables $VV$ for use in $S$. The behavior of the block command is the behavior of $S$ started with arbitrary values for $VV$.

### *1.5. Expressions*

*Expressions* are fairly standard and follow this grammar, where $\oplus$ denotes any binary operator shown in Fig. 1:

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Expr} \oplus \mathit{Expr} \\
 & | & \neg\ \mathit{Expr} \\
 & | & \mathit{Atom} \\
\mathit{Atom} & ::= & \mathit{Literal} \\
 & | & \mathit{Designator} \\
 & | & \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
 & | & \textbf{old}\ (\ \mathit{Expr}\ ) \\
 & | & \mathit{Quantification} \\
\mathit{Literal} & ::= & \textbf{false} \mid \textbf{true} \mid \textbf{null} \mid 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{Quantification} & ::= & (\ \mathit{Quantor}\ \mathit{IdType}^+ \bullet\ \mathit{Expr}\ ) \\
\mathit{Quantor} & ::= & \forall \mid \exists
\end{array}
$$

Unary and binary operators are given in Fig. 1. Each line shows the supported type signatures of the operators and common names for the operations. The figure also describes BoogiePL$^\flat$'s precedence rules. Each box holds operators with the same precedence. Operators in higher boxes have higher precedence than operators in lower boxes. For example, $a + b * c$ means $(a + (b * c))$, as usual. Implication is right associative. The other logical operators are associative, but associate only with themselves. All other operators are left associative. Although we do not show them in the grammar, we also allow expressions to contain parentheses, which can be used to override operator precedence.

The literals **false** and **true** have type **bool**, the literal **null** has type **ref**, and the integer literals have type **int**.

| Operator | Type | Description |
|---|---|---|
| $\neg$ | $: \mathbf{bool} \to \mathbf{bool}$ | logical negation |
| $*$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ | multiplication |
| $/$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ | integer division |
| $\%$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ | integer modulo |
| $+$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ | addition |
| $-$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ | subtraction |
| $<$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{bool}$ | arithmetic less-than |
| $\leqslant$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{bool}$ | arithmetic at-most |
| $\geqslant$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{bool}$ | arithmetic at-least |
| $>$ | $: \mathbf{int} \times \mathbf{int} \to \mathbf{bool}$ | arithmetic greater-than |
| $<:$ | $: \mathbf{name} \times \mathbf{name} \to \mathbf{bool}$ | partial order on names |
| $=$ | $: T \times T \to \mathbf{bool}$ | equality |
| $\neq$ | $: T \times T \to \mathbf{bool}$ | disequality |
| $\wedge$ | $: \mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$ | logical conjunction |
| $\vee$ | $: \mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$ | logical disjunction |
| $\Rightarrow$ | $: \mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$ | logical implication |
| $\Leftrightarrow$ | $: \mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$ | logical equivalence |

**Figure 1.** BoogiePL$^\flat$ operators, their types, and syntactic precedence.

In the function-application expression $f(EE)$, $f$ must refer to a function. The number of expressions in the list $EE$ must equal the number of arguments of $f$, and the types of the expressions in $EE$ must be assignable to the types of the corresponding arguments of $f$. The function-application expression has the same type as the result type of $f$.

The expression $\mathbf{old}(E)$ is allowed to appear only in $\mathbf{ensures}$ clauses and procedure implementations. If it appears in code, $E$ must only refer to variables that are in scope in the procedure's preconditions; more precisely, global variables, in-parameters, and quantifier-bound variables can be mentioned, but out-parameters and the implementation's local variables cannot. The expression denotes the value of $E$ on entry to the procedure.

The quantifier expression $( \mathcal{Q}\, w \bullet\ E\ )$, where $\mathcal{Q}$ is either $\forall$ or $\exists$, defines the identifiers in $w$ as bound variables that can be used in $E$. The type of a quantifier expression is $\mathbf{bool}$. The expression denotes the corresponding logical quantifier.

### *1.6. Weakest Preconditions*

The semantics of the commands in our simple language is defined by weakest preconditions [18,65].

The *weakest precondition* of a command $S$ and predicate $Q$ on the post-state of $S$, denoted by $wp[\![S, Q]\!]$, is a predicate on the pre-state of $S$ that characterizes the set of all states such that execution of $S$ begun in any of those states does not go wrong, and if it terminates successfully, terminates in $Q$.

We define the following:

$$
\begin{array}{lcl}
wp[\![\textbf{assert}\ E,\ Q]\!] & = & E \land Q \\
wp[\![\textbf{assume}\ E,\ Q]\!] & = & E \Rightarrow Q \\
wp[\![\textbf{havoc}\ xx,\ Q]\!] & = & (\forall\, xx \bullet\ Q\ ) \\
wp[\![x := E,\ Q]\!] & = & Q[E/x] \\
wp[\![S\ ;\ T,\ Q]\!] & = & wp[\![S,\ wp[\![T,Q]\!]]\!]) \\
wp[\![S\ []\ T,\ Q]\!] & = & wp[\![S,Q]\!] \land wp[\![T,Q]\!] \\
wp[\![\{\,VV\ S\},\ Q]\!] & = & (\forall\, vv \bullet\ wp[\![S,Q]\!]\ )
\end{array}
$$

where $vv$ denotes the list of variables declared in $VV$, and where we understand a quantification with an empty list of bound variables to be just the body of the quantification. The semantics of map assignment is defined in Section 1.8 and the semantics of procedure calls is defined below.

The translation function $Q[E/x]$ denotes the capture-avoiding substitution of $E$ for $x$ in $Q$ that keeps all **old** subexpressions intact. For example, if $Q$ is

$$x < \textbf{old}(x) \land (\forall\, x \bullet\ 0 \leqslant g(x)\ ) \land (\forall\, y \bullet\ f(x,y) \Rightarrow g(x)\ )$$

then $Q[y+1/x]$ is

$$y + 1 < \textbf{old}(x) \land (\forall\, x \bullet\ 0 \leqslant g(x)\ ) \land (\forall\, z \bullet\ f(y+1,z) \Rightarrow g(y+1)\ )$$

where, to avoid variable capture, the substitution operation renamed the bound variable $y$ to a fresh variable $z$.

The definition of assert says that the legal pre-states of **assert** $E$ are those in which both $E$ and $Q$ hold. Here and in Spec$^\flat$, programmers use **assert** $E$ to claim that the condition $E$ holds, and a program verifier must verify that claim.

The definition of assume says that the legal pre-states of **assume** $E$ are those in which either $E$ does not hold or $Q$ already holds. Here and in Spec$^\flat$, programmers use **assume** $E$ to express the fact that they care only about those executions where $E$ holds, and a program verifier is then allowed to use $E$ as an assumption.

The definition of **havoc** $xx$ says that $Q$ has to hold for all possible values of $xx$.

The assignment says that in order for $Q$ to hold after the assignment, $Q$ with $x$ replaced by $E$ must hold before it.

Sequential composition and choice correspond to functional composition and conjunction, respectively.

The meaning of the block command $\{\textbf{var}\ x\colon T;\ C\}$ is defined in terms of the meaning of its embedded command $C$ by universally quantifying over all possible initial values of $x$. Note that the block command is equivalent to **havoc** $x\ ;\ C$, but the block command also introduces the scope of $x$.

Here are some example derivations for $wp$, where we have simplified some of the right-hand sides:

$$
\begin{array}{lcl}
wp[\![\textbf{assert}\ 1 < x,\ Q]\!] & = & 1 < x \land Q \\
wp[\![\textbf{assert true},\ Q]\!] & = & Q \\
wp[\![i := i + 1,\ i \leqslant 1]\!] & = & i \leqslant 0 \\
wp[\![\textbf{assume}\ y = x + 1,\ y = 5]\!] & = & y = x + 1 \Rightarrow y = 5 \\
wp[\![\textbf{assume false},\ Q]\!] & = & \mathit{true} \\
wp[\![\textbf{assert}\ P\ ;\ \textbf{assume}\ P,\ Q]\!] & = & P \land (P \Rightarrow Q)
\end{array}
$$

*Shorthand Notations* If a call command has no out-parameters, we write it simply as **call** $P(EE)$.

In the definition of procedure calls below, it will be convenient to use simultaneous assignments, which we write as $xx := EE$, where $xx$ is a list of distinct writable variables, $EE$ is a list of expressions of the same length, and each expression in $EE$ is assignable to the type of the corresponding variable in $xx$. The values of the variables after executing an assignment command $xx := EE$ equal the values of the corresponding expressions before executing it. Simultaneous assignments can be defined in terms of block commands and assignments: a block command introduces temporary variables for the variables in $xx$, then assigns, in sequence, each of the expressions in $EE$ to the temporary variables, and finally assigns each of the temporary variables to $xx$. The $wp$ of simultaneous assignment then becomes:

$$wp[\![xx := EE,\ Q]\!] \;=\; Q[EE/xx]$$

where $[EE/xx]$ denotes simultaneous substitution.

*Procedures* To define the semantics of procedure calls and implementations, we refer to the names in the following schema:

$$\begin{array}{l}\mathbf{procedure}\ P(AA)\ \mathbf{returns}\ (RR);\\ \quad \mathbf{requires}\ Pre;\ \ \mathbf{modifies}\ gg;\ \ \mathbf{ensures}\ Post;\end{array}$$

The semantics of a procedure call **call** $xx := P(EE)$ is defined to be the semantics of the following command:

$$\begin{array}{l}\{\ \mathbf{var}\ AA;\ \mathbf{var}\ RR;\ \mathbf{var}\ HH;\\ \quad aa := EE;\\ \quad \mathbf{assert}\ Pre;\\ \quad hh := gg;\\ \quad \mathbf{havoc}\ gg;\\ \quad \mathbf{assume}\ StripOld[\![ReplaceOld[\![Post, gg, hh]\!]]\!];\\ \quad xx := rr\\ \}\end{array} \tag{0}$$

where $HH$ is a list of fresh variable declarations corresponding to the global variables $gg$, and where we use $aa$, $rr$, and $hh$ to denote the lists of identifiers introduced by $AA$, $RR$, and $HH$, respectively. Here and in the sequel, we are sloppy with the exact syntax of lists, as in showing just one **var** keyword in front of the list $AA$.

The definition of procedure call introduces local variables for the formal parameters $AA$ and $RR$, and introduces a fresh variable in $HH$ for every global variable mentioned in $gg$. The definition then evaluates the actual in-parameters and assigns these to the formals. The definition then requires that the caller has established the precondition of $P$. The **havoc** command destroys all knowledge about modified global variables; the assignment $hh := gg$ captures the previous values of $gg$. The caller can then assume that the postcondition has been established, where in the postcondition we first handle **old** expressions: for each variable $g$ in $gg$, the translation function $ReplaceOld[\![Post, gg, hh]\!]$ replaces every occurrence of $g$ nested inside an **old** expression within $Post$ by $g$'s corresponding variable in $hh$; translation function $StripOld[\![Q]\!]$ replaces every subexpres-

$$
\begin{array}{l}
\textbf{function}\ f(\textbf{int})\ \textbf{returns}\ (\textbf{int})\ ; \\
\textbf{axiom}\ (\forall\, k\colon \textbf{int} \bullet\ 0 \leqslant k\ \Rightarrow\ 2 * k \leqslant f(k)\ )\ ; \\
\textbf{var}\ x\colon \textbf{int}\ ; \\
\textbf{procedure}\ \mathit{Inc}(n\colon \textbf{int})\ \textbf{returns}\ (r\colon \textbf{int})\ ; \\
\quad \textbf{requires}\ 0 \leqslant n\ ; \\
\quad \textbf{modifies}\ x\ ; \\
\quad \textbf{ensures}\ \textbf{old}(x) \leqslant x \wedge r = \textbf{old}(x)\ ; \\
\textbf{implementation}\ \mathit{Inc}(n\colon \textbf{int})\ \textbf{returns}\ (r\colon \textbf{int}) \\
\{\ r := x\ ;\ x := x + f(n)\ \}
\end{array}
$$

**Figure 2.** An example BoogiePL$^\flat$ program, showing the declaration of a function $f$, an axiom that constrains $f$, a global variable $x$, a procedure $Inc$ that is specified to operate on $x$, and an implementation of $Inc$.

sion $\textbf{old}(E)$ in $Q$ by $E$ (we omit the formal definitions of $\mathit{ReplaceOld}$ and $\mathit{StripOld}$). Finally, the definition of the call assigns the formal out-parameters to the actuals.

This definition is correct only if $AA$ and $RR$ do not capture variables used in $EE$. If they do, we have to introduce fresh variables in $AA$ and $RR$ and consistently rename the uses of the formal in- and out-parameters in $\mathit{Pre}$ and $\mathit{Post}$ before unfolding the definition above.

An implementation

$$
\begin{array}{l}
\textbf{implementation}\ P(AA)\ \textbf{returns}\ (RR) \\
\quad \mathit{Body}
\end{array}
$$

of procedure $P$ is *valid* if it obeys the procedure's specification, under the proviso of the mathematical theory (that is, the conjunction of the axioms), here called $MT$:

$$
\begin{array}{l}
\mathit{valid}(P, \mathit{Body}) = \\
\quad \mathit{StripTypes}[\![ MT\ \Rightarrow\ \mathit{StripOld}[\![\ \mathit{wp}[\![ \{\ \begin{array}[t]{l} \textbf{var}\ AA;\ \textbf{var}\ RR; \\ \textbf{assume}\ \mathit{Pre}; \\ \mathit{Body}; \\ \textbf{assert}\ \mathit{Post}; \end{array} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \},\ \ \mathit{true}\ ]\!]]\!]]\!]
\end{array}
\tag{1}
$$

The application of $\mathit{wp}$ produces a predicate on the pre-state of the procedure. In that state, $\textbf{old}(E)$ means just $E$, so we apply the translation function $\mathit{StripOld}$. The translation function $\mathit{StripTypes}$ erases the types of all quantifier-bound variables (we omit the formal definition).

Note that, compared to the call, the roles of the assert and assume commands are reversed here. Also, note that the modifies clause $gg$ need not be verified, since it is already syntactically checked ($\mathit{Body}$ is allowed to assign only to writable variables).

We say that a BoogiePL$^\flat$ program is *correct* if all its procedure implementations are valid. Note that this verification technique is *modular*, since it verifies each implementation separately.

### *1.7. Example*

Consider the BoogiePL$^\flat$ program in Fig. 2. The mathematical theory, $MT$ in (1), of this

program is its one axiom. For the implementation of $Inc$, the command to which $wp$ is applied in (1) is:

$$\begin{array}{l} \{\ \textbf{var}\ n{:}\,\textbf{int}\ ;\ \ \textbf{var}\ r{:}\,\textbf{int}\ ; \\ \quad \textbf{assume}\ 0 \leqslant n\ ; \\ \quad \{\ r := x\ ;\ \ x := x + f(n)\ \}\ ; \\ \quad \textbf{assert}\ \textbf{old}(x) \leqslant x \wedge r = \textbf{old}(x) \\ \} \end{array}$$

The $wp$ of this command with respect to $true$ is:

$$(\forall\, n, r \bullet\ 0 \leqslant n \ \Rightarrow\ \textbf{old}(x) \leqslant x + f(n) \wedge x = \textbf{old}(x) \wedge \mathit{true}\ )$$

Applying $StripOld$ to this formula yields:

$$(\forall\, n, r \bullet\ 0 \leqslant n \ \Rightarrow\ x \leqslant x + f(n) \wedge x = x \wedge \mathit{true}\ )$$

So, the verification condition generated for the program in Fig. 2 is:

$$\begin{array}{l} (\forall\, k \bullet\ 0 \leqslant k \ \Rightarrow\ 2 * k \leqslant f(k)\ ) \ \Rightarrow \\ \quad (\forall\, n, r \bullet\ 0 \leqslant n \ \Rightarrow\ x \leqslant x + f(n) \wedge x = x \wedge \mathit{true}\ ) \end{array}$$

This is a valid formula, which an SMT solver like Simplify [12] easily verifies, so the program is correct.

By definition (0), the semantics of a call command:

$$\textbf{call}\ z := \mathit{Inc}(17)$$

is given by the following command:

$$\begin{array}{l} \{\ \textbf{var}\ n{:}\,\textbf{int}\ ;\ \ \textbf{var}\ r{:}\,\textbf{int}\ ;\ \ \textbf{var}\ \mathit{old_x}{:}\,\textbf{int}\ ; \\ \quad n := 17\ ; \\ \quad \textbf{assert}\ 0 \leqslant n\ ; \\ \quad \mathit{old_x} := x\ ; \\ \quad \textbf{havoc}\ x\ ; \\ \quad \textbf{assume}\ \mathit{old_x} \leqslant x \wedge r = \mathit{old_x}\ ; \\ \quad z := r \\ \} \end{array}$$

### *1.8. Targeting a Theorem Prover*

The correctness of an implementation $Body$ of a procedure $P$ is verified by passing $valid(P, Body)$ to an theorem prover, like a Satisfiability Modulo Theories (SMT) solver. Modern SMT solvers, like Simplify [12], are particularly well suited for automatic verification. First, they require no user interaction and can thus be used as a push-button technology. Second, they are refutation based, that is, they may produce counterexamples in case a property can't be satisfied, and those counterexamples can be used for error reporting. Third, their heuristics are tuned for software verification.

These SMT solvers are typically built around Nelson-Oppen cooperating decision procedures [64,66]. They all provide decision procedures for congruence closure (uninterpreted function symbols and equality), linear arithmetic, and quantifiers. Some also

provide partial orders, maps, and other theories. In case a targeted theorem prover does not support a BoogiePL$^\flat$ defined operator, we have to add proper axioms. Simplify, for instance, does not have a built-in decision procedure for maps. Consequently, the verification-condition generator for Simplify has to axiomatize operators for map select and map update.

For instance, the verification-condition generator for Simplify replaces every map select expression $A[E]$ by the term $select(A, E)$, and replaces every map update $A[E] := F$ by $A := store(A, E, F)$. Simplify is untyped, so it suffices to add one axiom for $select$ and $store$ to the theorem prover's background axioms:

$$(\forall\, m, i, j, v \bullet \;\; (i = j \;\Rightarrow\; select(store(m, i, v), j) = v) \wedge \\ (i \neq j \;\Rightarrow\; select(store(m, i, v), j) = select(m, j))\;)$$

Of course, arities of function symbols have to be respected in Simplify, that is, we need different function symbols and axioms to support maps of different arities.

## 2. An Object-oriented Programming Language

This section defines Spec$^\flat$, an object-oriented programming language. Spec$^\flat$ is a core of Spec$^\sharp$ [6]. Like the modern object-oriented languages Java, C$^\sharp$, Eiffel, and Modula-3, Spec$^\flat$ has object references, classes, subclasses with single inheritance, methods with dynamic dispatch, and co-variant arrays. Spec$^\flat$ excludes features like interfaces, multiple inheritance, structs, delegates, generics, static members, once functions, abstract methods, properties, events, iterators, overloading, boxing, and visibility modifiers.

Figure 3 shows an example Spec$^\flat$ program.

We give the semantics of Spec$^\flat$ in terms of a translation into the procedural language BoogiePL$^\flat$.

### 2.0. Programs, Classes, and Members

At the top level, a Spec$^\flat$ program is a set of classes.

$$Prog \;::=\; Class^*$$

A *class* defines an object type and provides its implementation. A class has a name (an identifier), a superclass, and a set of member declarations.

$$\begin{aligned} Class \;&::=\; \textbf{class}\; Id \;:\; ObjectType \;\{\; Member^* \;\} \\ ObjectType \;&::=\; \textbf{object} \mid Id \end{aligned}$$

Each class derives from a single existing class, its *immediate superclass*. The declared classes form a single-inheritance subtype hierarchy rooted at the built-in object type **object**. As a shorthand, we allow ": $ObjectType$" to be omitted; $ObjectType$ then defaults to **object**. The values of object types are called *objects* and consist of the special value **null** and of *references* to a suite of class members (fields, invariants, and methods). Every reference has a built-in readonly field called $Type$, which returns the run-time type of the reference, represented as an object.

In addition to object types, we consider the Spec$^\flat$ types booleans, and integers, and one-dimensional arrays.

```
class Cell {
  int x;

  Cell(int i)
    ensures x = i;
  { x = i; }

  virtual int Get()
    ensures result = x;
  { return x; }

  virtual void Set(int i)
    modifies this.*;
    ensures x = i;
  { x = i; }

  void IncBy(int i)
    modifies this.*;
    ensures x = old(x) + i;
  { int t = Get(); Set(t + i); }
}
```

```
class BackupCell : Cell {
  int b;

  BackupCell(int i)
    ensures b = i && x = i;
  { base(i); b = i; }

  override void Set(int i)
    ensures b = old(x);
  { b = x; base.Set(i); }

  virtual int GetBackup()
    ensures result = b;
  { return b; }

  void Rollback()
    modifies x;
    ensures x = b;
  { x = b; }
}
```

**Figure 3.** Two example classes written in Spec$^\flat$. Class *Cell* represents a single storage location. The subclass *BackupCell* additionally maintains a recent history of the contents of that storage location.

$$Type \quad ::= \quad \textbf{bool} \mid \textbf{int} \mid ObjectType \mid Type\,[\,]$$

Arrays are *references* to sequences of values. Each array type is a subtype of **object**. We refer to object types and array types as *reference types*. The types respect *polymorphic subtyping*, that is, if $T$ is a subtype of $S$, then an expression of type $T$ can be assigned to a designator of type $S$ (but not vice versa, unless $T$ and $S$ are the same type). Our array types are *co-variant* in their element type. For example, the type $Point[\,]$ is a subtype of **object**$[\,]$, provided *Point* is a subtype of **object**. Arrays support the usual indexing lookup and update operations; they also have a built-in readonly field called *Length*.

Class members can be fields and methods.

$$Member \quad ::= \quad Field \mid Method$$

A *field* is an instance variable, that is, each instance of the class has its own copy of the variable.

$$Field \quad ::= \quad FieldModifier^? \; Type \; Id \; ;$$

Modifiers for fields will be introduced later.

Spec$^\flat$ supports usual scoping rules. For simplicity, we assume here that the fields declared in a class are distinct from other fields declared in the class and its superclasses.

A *method* is a name for a parameterized operation on the state space. A method can be invoked, passing a fixed number of values as parameters. Every method declaration belongs to some class. Syntactically, Spec$^\flat$ distinguishes three kinds of methods: *con-*

*structors*, *non-virtual methods*, and *virtual methods*. When a method includes a **virtual** or **override** modifier, that method is said to be a virtual method; otherwise, the method is said to be a non-virtual method. A constructor declaration looks like a non-virtual method declaration, but it is named with the name of the class in which it is defined and it has no result type.

A non-virtual method must be distinct from all methods declared in the class and its superclasses. The implementation of a non-virtual method is the same whether the method is invoked on an instance of the class in which it is declared or on an instance of a derived class.

A virtual method declared with **virtual** must be distinct from all other methods declared in the class and its superclasses. In contrast to non-virtual methods, the implementation of a virtual method can be overridden in derived classes. The declaration of the method override, indicated by the **override** keyword, must have the same signature as the overridden method; it provides a new implementation of the overridden method.

Every class has a constructor. It is only used in the creation of an object of the class. Constructors are implicitly called by class instance creation expressions (**new**) and by base calls inside constructors. For simplicity, we restrict classes to have just one constructor. If a class declares no constructor, then a default constructor with no parameters is provided, which simply calls the superclass constructor with no parameters (which is an error if the superclass does have a parameterless constructor). If a constructor is given, the first statement of its body must be a call to the superclass constructor.

$$
\begin{array}{rcl}
\mathit{Method} & ::= & \mathit{NonConstrPrefix}^{?} \\
 & & \mathit{Id}\ (\mathit{TypeId}^{*})\ \mathit{Specification}^{*}\ \mathit{Block} \\
\mathit{NonConstrPrefix} & ::= & \mathit{MethodModifier}^{?}\ \mathit{ReturnType} \\
\mathit{MethodModifier} & ::= & \mathbf{virtual} \mid \mathbf{override} \\
\mathit{ReturnType} & ::= & \mathit{Type} \mid \mathbf{void} \\
\mathit{TypeId} & ::= & \mathit{Type}\ \mathit{Id} \\
\mathit{Specification} & ::= & \mathbf{requires}\ \mathit{Expr}\ ; \\
 & \mid & \mathbf{modifies}\ \mathit{ModDesignator}^{+}\ ; \\
 & \mid & \mathbf{ensures}\ \mathit{Expr}\ ; \\
\mathit{ModDesignator} & ::= & \mathit{ODesignator}\ \mathit{ModSuffix} \\
\mathit{ODesignator} & ::= & \mathbf{this} \mid \mathit{Designator} \\
\mathit{ModSuffix} & ::= & .\ \mathit{Id} \mid\ .\ * \mid [\, * \,]
\end{array}
$$

In addition to the explicitly declared parameters, each method takes an implicit *receiver* parameter referred to by the keyword **this**. If a method has no return value, its return type is specified as **void**.

The procedure specification consists of a number of **requires**, **modifies**, and **ensures** clauses, which like in BoogiePL$^{\flat}$ introduce preconditions, modifies clauses, and postconditions. The expressions in the pre- and postconditions must be of type **bool**. The parameters, including **this**, are in scope in the specification, except that **this** is not available in the precondition of constructors. Postconditions can also mention **old** and **fresh** expressions (explained below) and, for non-void, non-constructor methods, the keyword **result**, which denotes the return value. The pre- and postconditions are not allowed to contain allocation and call expressions (explained below). The modifies clause must not list the designator expression $E.\mathit{Type}$ or $E.\mathit{Length}$, for any expression $E$, at the top level.

Each method has one implementation, consisting of a block statement. Unlike in BoogiePL$^\flat$, we cannot enforce modifies clauses in Spec$^\flat$ by syntactic restrictions. Instead, method implementations in Spec$^\flat$ need to be verified to satisfy their modifies clauses. The declared modifies clauses indicate a set of heap locations, which one gets by evaluating every modifies designator expression on entry to the method. A modifies designator of the form $E.f$ gives the license to modify the $f$ field of object $E$, $E.*$ gives the license to modify any field of object $E$, and $E[*]$ gives the license to modify array $E$ at any index. A method implementation also gets a blanket license to modify some other things, as we explain later.

Spec$^\flat$ supports *behavioral subtyping* [42,55,16], that is, whenever an object of static type $S$ is expected, any object of a subtype $T$ of $S$ can be used without invalidating the program's verification. A necessary condition for behavioral subtyping is the following: Consider a virtual method $m$ defined in $S$ (written $S^\circ m$) and an override of $m$ defined in a subclass $T$ (written $T^\circ m$); then, $T^\circ m$ can only weaken $S^\circ m$'s precondition and only strengthen $S^\circ m$'s postcondition. In this paper, we consider only the strengthening of postconditions, which override $T^\circ m$ can specify by providing additional **ensures** clauses. These are then conjoined with the postconditions of the overridden method, as the translation into BoogiePL$^\flat$ will make explicit.

### 2.1. Statements

Statements in Spec$^\flat$ follow this grammar:

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathit{Block} \\
 & | & \mathit{Type}\ \mathit{Id}\ ; \\
 & | & \mathit{Type}\ \mathit{Id}\ =\ \mathit{Expr}\ ; \\
 & | & \mathbf{assert}\ \mathit{Expr}\ ; \\
 & | & \mathbf{assume}\ \mathit{Expr}\ ; \\
 & | & \mathit{Designator}\ =\ \mathit{Expr}\ ; \\
 & | & \mathit{Call} \\
 & | & \mathit{IfStmt} \\
 & | & \mathit{WhileStmt} \\
 & | & \mathbf{return}\ \mathit{Expr}\ ; \\
\mathit{Block} & ::= & \{\ \mathit{Stmt}^*\ \} \\
\mathit{IfStmt} & ::= & \mathbf{if}\ (\ \mathit{Expr}\ )\ \mathit{Stmt}\ \mathit{ElseStmt}^? \\
\mathit{ElseStmt} & ::= & \mathbf{else}\ \mathit{Stmt} \\
\mathit{WhileStmt} & ::= & \mathbf{while}\ (\ \mathit{Expr}\ )\ \mathit{Invariant}^?\ \mathit{Block} \\
\mathit{Invariant} & ::= & \mathbf{invariant}\ \mathit{Expr}\ ;
\end{array}
$$

If a statement attempts to evaluate an undefined expression (like $x/y$ when $y$ evaluates to 0), the statement *goes wrong*, which is an unrecoverable error.

The block statement consists of a sequence of statements, which are executed in order. The declaration statement $T\ x$; introduces a local variable $x$ whose scope goes from the declaration until the end of the enclosing block. As usual, $x$ must be distinct from any other variable introduced among the statements, but, unlike in BoogiePL$^\flat$, a variable in Spec$^\flat$ must not hide another local variable $x$ in an outer scope. If a local variable hides a field in scope, then the field has to be accessed via **this** explicitly.

The statement $T\ x =\ E$; is simply a shorthand for $T\ x;\ x =\ E$;.

Syntax and semantics of **assert** and **assume** are the same as for BoogiePL$^\flat$, except that the Spec$^\flat$ statements also check that their expressions are defined. Embedded expressions must not contain call or allocation expressions. In Spec$^\flat$, the assume statement introduces an assumption that is used, but not validated, by the program verifier; the assumption can instead be validated at run time.

In the assignment statement $d = E$;, the type of $E$ must be a subtype of the type of $d$. If $d$ is of the form $O.f$, then $f$ must not be the built-in readonly fields $\mathit{Type}$ or $\mathit{Length}$. The statement goes wrong if $d$ or $E$ is not defined; otherwise, it assigns the value of $E$ to $d$. More specifically, if $d$ is of the form $O.f$, the statement updates the heap so that field $f$ of object $O$ becomes $E$; if $d$ is of the form $A[F]$, the statement updates the heap so that element $F$ of array $A$ becomes $E$.

The call statement is explained in Section 2.2. Only constructors (called via $\mathit{Base}$) and methods with a void return type can be used as $\mathit{Call}$ statements.

In the conditional statement **if** $(E)$ $S$ **else** $T$, expression $E$ must be of type **bool**. The statement goes wrong if $E$ is not defined. Otherwise, if $E$ evaluates to **true**, the statement executes $S$, and if $E$ evaluates to **false**, the statement executes $T$. If "**else** $S$" is omitted, $S$ defaults to $\{\,\}$. Parsing of conditional statements is ambiguous; we resolve any ambiguity of parsing **else** statements by associating them to the rightmost (innermost) **if**.

In the while loop **while** $(E)$ **invariant** $J$; $\mathit{Body}$, the condition $E$ and loop invariant $J$ must be of type **bool**. Furthermore, $J$ must not contain call or allocation expressions. As we explained in Section 1.3, the statement goes wrong if $J$ is not defined, if $J$ does not evaluate to **true**, or if $E$ is not defined. Otherwise, if $E$ evaluates to **true**, then $\mathit{Body}$ is executed, after which the entire while loop is executed again (including the evaluation of the loop invariant). If $E$ evaluates to **false**, the execution of the while loop terminates.

In the return statement **return** $E$;, the type of $E$ must be a subtype of the method's return type. The return statement is allowed only as the last statement of a method implementation. The statement goes wrong if $E$ is not defined; otherwise, it returns the value of $E$ to the method's caller.

### 2.2. *Expressions*

Spec$^\flat$ expressions follow the grammar in Fig. 4.

Spec$^\flat$ shares most of its operators with BoogiePL$^\flat$. Except as noted here, the shared operators have the same typing, precedence, and meaning. In Spec$^\flat$, division and modulo are defined only for non-zero divisors. For the relational operators $=$ and $\neq$, the types of the operands must be *compatible*; that is, the type of one operand must be a subtype of the type of the other.

Instead of the logical operators $\Rightarrow$, $\wedge$, and $\vee$ in BoogiePL$^\flat$, Spec$^\flat$ defines the corresponding *short-circuit* versions of these operators, written $\stackrel{\cdot}{\Longrightarrow}$, $\&\&$, and $||$, respectively. Short-circuiting means that if the left-hand operand is defined and evaluates to a value that determines the result of the operator (**false**, **false**, and **true**, respectively), then the expressions are defined regardless of whether or not the right-hand operand is defined.

Spec$^\flat$ also adds the binary **is** operator, whose precedence lies between those of $=$ and $\&\&$. The precedence of cast expressions, where the prefix "$(T)$" is like a unary operator, is just higher than $\otimes$. Although we do not show them in the grammar, we

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Expr} \ \otimes \ \mathit{Expr} \\
& | & \neg\ \mathit{Expr} \\
& | & \mathit{Atom} \\
\otimes & ::= & * \mid / \mid \% \mid + \mid - \mid < \mid \leqslant \mid \geqslant \mid > \mid = \mid \neq \\
& | & \&\& \mid || \mid \dot{\Longrightarrow} \\
\mathit{Atom} & ::= & \mathbf{this} \mid \mathbf{result} \\
& | & \mathit{Literal} \\
& | & \mathit{Designator} \\
& | & \mathit{Call} \\
& | & \mathit{Allocation} \\
& | & \mathit{Expr}\ \mathbf{is}\ \mathit{Type} \\
& | & (\ \mathit{Type}\ )\ \mathit{Expr} \\
& | & \mathbf{old}\ (\ \mathit{Expr}\ ) \\
& | & \mathbf{fresh}\ (\ \mathit{Expr}\ ) \\
& | & \mathit{Quantification} \\
\mathit{Literal} & ::= & \mathbf{false} \mid \mathbf{true} \mid \mathbf{null} \mid 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{Designator} & ::= & \mathit{Id} \\
& | & \mathit{Expr}\ .\ \mathit{Id} \\
& | & \mathit{Expr}\ [\ \mathit{Expr}\ ] \\
\mathit{Call} & ::= & \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
& | & \mathit{Expr}\ .\ \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
& | & \mathbf{base}\ (\ \mathit{Expr}^*\ ) \\
& | & \mathbf{base}\ .\ \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
\mathit{Allocation} & ::= & \mathbf{new}\ \mathit{ObjectType}\ (\ \mathit{Expr}^*\ ) \\
& | & \mathbf{new}\ \mathit{Type}\ [\ \mathit{Expr}\ ] \\
\mathit{Quantification} & ::= & \mathit{Quantor}\ \{\ \mathit{Binding}\ ;\ \mathit{Expr}\ \} \\
\mathit{Quantor} & ::= & \mathbf{forall} \mid \mathbf{exists} \\
\mathit{Binding} & ::= & \mathbf{int}\ \mathit{Id}\ \mathbf{in}\ (\ \mathit{Expr}\ :\ \mathit{Expr}\ )
\end{array}
$$

**Figure 4.** The grammar of Spec$^\flat$ expressions.

also allow expressions to contain parentheses, which can be used to override operator precedence.

The type of the expression **this** is the enclosing class. The keyword **result** is allowed to appear only in **ensures** clauses of non-void, non-constructor methods; its type is the method's return type, and its value is the method's return value. The type of **null** is any reference type. The type of boolean literals is **bool**, the type of integer literals is **int**.

Three forms of *designators* are distinguished. In the first form, if $Id$ does not mention a variable, then it is a synonym for **this**.$Id$. The type of the expression $Id$ is the type of the variable $Id$. In the second form, $E.f$, $E$ must be of a reference type, call it $T$, and $f$ must be *Type*, *Length* (if $E$ is of an array type), or a field declared in class $T$ or a superclass thereof. The expression is defined only if $E$ evaluates to a non-null value. The type of $E.\mathit{Type}$ is **object**, the type of $E.\mathit{Length}$ is **int**, and otherwise the type of $E.f$ is the type of $f$. In the third form, $E[F]$, $E$ must be of an array type and $F$ of type integer. The expression is defined only if $E$ evaluates to a non-null value, $F$ evaluates

to a non-negative integer that is less than $E.Length$. The type of the expression is the element type of $E$.

Spec$^\flat$ supports four forms of *call expressions*. All legal call expressions resolve to some method. The types of the expressions in list $EE$ must be subtypes of the types of the respective formal parameters of the callee. If the callee is a constructor or has a void return type, then the call is allowed only as a statement; otherwise, the type of the call expression is the return type of the callee.

The first form of the call expression, $m(EE)$, is a shorthand for $\mathbf{this}.m(EE)$.

In the second form, $E.m(EE)$, $E$ must be of an object type, call it $T$, and $m$ must name a non-constructor method in $T$ or a superclass thereof. The call expression is defined only if $E$ evaluates to a non-null value. The expression binds the formal receiver parameter $\mathbf{this}$ of $m$ to the value of $E$ and binds the formal parameters of $m$ to the values of $EE$. Then, if any precondition of $m$ evaluates to **false**, the call statement goes wrong. The evaluation of the call expression proceeds by transferring control to the method's implementation, upon return of which the result value becomes the value of the call expression. If $m$ is a virtual method, then the implementation invoked is the one found in the most derived supertype of the run-time type of $E$.

The third form, $\mathbf{base}.m(EE)$ where $m$ must denote a virtual method, is allowed only in overrides of $m$. It is treated like the call $\mathbf{this}.m(EE)$, except that control transfers to the implementation found in the most derived supertype of the immediate superclass of the enclosing class.

The fourth form, $\mathbf{base}(EE)$, is allowed only as the first statement in constructors. It calls the constructor of the immediate superclass.

Two forms of *allocation expressions* are supported. The expression $\mathbf{new}\ C(EE)$, where $C$ must name a class, has type $C$. It allocates a new object $c$ of run-time type $C$ with all of $c$'s fields set to zero-equivalent values. Next, it calls $C's$ constructor with $c$ as the receiver and $EE$ as its actual parameters. All constraints of method calls have to be obeyed. Upon return of the constructor, $c$ is the result of the expression. The expression $\mathbf{new}\ T[F]$, where $T$ must be a type and $F$ must be of type **int**, has type $T[\,]$. It allocates and returns a new array $a$ of run-time type $T[\,]$ and of length $F$. All array elements of $a$ have zero-equivalent values. The statement goes wrong if either $F$ is not defined or if $F$ evaluates to a negative integer.

For the type test expression $E$ **is** $T$ and cast expression $(T)E$, $T$ must be a reference type and the type of $E$ must be compatible with $T$. The type of $E$ **is** $T$ is **bool**, the type of $(T)E$ is $T$. The type test expression evaluates to **true** if $E$ evaluates to a non-null reference whose run-time type is a subtype of $T$. The cast expression is defined only if $E$ evaluates to **null** or if $E$ **is** $T$ would evaluate to **true**, and it returns the value of $E$.

The expressions $\mathbf{old}(E)$ and $\mathbf{fresh}(E)$ are allowed to appear only in **ensures** clauses. (Note, unlike BoogiePL$^\flat$, Spec$^\flat$ does not allow **old** expressions in code.) The type of $\mathbf{old}(E)$ is the type of $E$ and the type of $\mathbf{fresh}(E)$ is **bool**. The former returns the value of $E$ evaluated on entry to the method; the latter returns **true** if $E$ denotes an object that was not yet allocated on method entry.

In a quantifying expression $\mathbf{forall}\{\mathbf{int}\ x\ \mathbf{in}\ (M : N);\ E\}$, $M$ and $N$ must have type **int** and the expression $E$ must have type **bool**. The newly introduced $x$ is in scope in $E$, but not in $M$ or $N$. Expressions $M$, $N$, and $E$ must not include call or allocation expressions. The quantifying expression has type **bool**. It is defined and returns

**true**, respectively, if $E$ is defined and evaluates to **true**, respectively, for every value of $x$ satisfying $N \leqslant x < M$. Existential quantification is defined in terms of universal quantification in the usual way.

### *2.3. Translating Spec$^\flat$ into BoogiePL$^\flat$*

We give the semantics of Spec$^\flat$ in terms of a translation into BoogiePL$^\flat$.

#### *2.3.0. Prelude*

The translation into BoogiePL$^\flat$ begins with the prelude described in this subsection. The prelude is specific to Spec$^\flat$, but independent of the particular program being translated.

*Axiomatizing the Type System* We map the Spec$^\flat$ types **bool** and **int** to the corresponding BoogiePL$^\flat$ types (we ignore the fact that Spec$^\flat$'s integers have a fixed size). We map all reference types in Spec$^\flat$ to the BoogiePL$^\flat$ type **ref**.

We introduce a name for each Spec$^\flat$ type. The names of the built-in types are:

$$\begin{array}{l}\mathbf{const}\ _bool\colon \mathbf{name}\ ;\\ \mathbf{const}\ _int\colon \mathbf{name}\ ;\\ \mathbf{const}\ object\colon \mathbf{name}\ ;\end{array}$$

Since the names are declared as constants of type **name**, BoogiePL$^\flat$ provides the implicit axiom that the type names are distinct.

Spec$^\flat$'s subtyping relation is captured by BoogiePL$^\flat$'s partial-order operator $<:$ and is specified via axioms. To tie $<:$ to type names, we introduce a function *superclass*:

$$\begin{array}{l}\mathbf{function}\ superclass(\mathbf{name})\ \mathbf{returns}\ (\mathbf{name})\ ;\\ \mathbf{axiom}\ (\forall\, T\colon \mathbf{name} \bullet\ T <: superclass(T)\ )\ ;\end{array}$$

The function *array* maps a type name $T$ to the name of an array type. Given the name of such an array type, function *elemType* gives back $T$.

$$\begin{array}{l}\mathbf{function}\ array(\mathbf{name})\ \mathbf{returns}\ (\mathbf{name})\ ;\\ \mathbf{function}\ elemType(\mathbf{name})\ \mathbf{returns}\ (\mathbf{name})\ ;\\ \mathbf{axiom}\ (\forall\, T\colon \mathbf{name} \bullet\ elemType(array(T)) = T\ )\ ;\end{array}$$

Array types are distinct and co-variant:

$$\begin{array}{l}\mathbf{axiom}\ (\forall\, T\colon \mathbf{name} \bullet\ array(T) <: object\ )\ ;\\ \mathbf{axiom}\ (\forall\, T\colon \mathbf{name},\ U\colon \mathbf{name} \bullet\ array(T) <: U\ \Rightarrow\\ \quad U = object \lor (U = array(elemType(U)) \land T <: elemType(U))\ )\ ;\end{array}$$

Fields are also declared to be of type **name**. We introduce a function that maps names of fields to the names of their declared types:

$$\mathbf{function}\ fieldType(\mathbf{name})\ \mathbf{returns}\ (\mathbf{name})\ ;$$

Function *type* returns the name of the run-time type of a non-null reference.

$$\mathbf{function}\ type(\mathbf{ref})\ \mathbf{returns}\ (\mathbf{name})\ ;$$

If $o$ has static type $T$ for a reference type $T$, then the static type system guarantees that $type(o) <: T$ holds.

*Storage Model* We model the heap as a map from references and field names to values.

$$\mathbf{var}\ \mathcal{H}\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any}\ ;$$

Our heap variable includes all references, allocated or not. We introduce a field $alloc$ to track whether or not a reference has been allocated. We refer to such a field as a *ghost field*, meaning that it is not explicitly represented in the Spec$^\flat$ program.

$$\mathbf{const}\ alloc\colon \mathbf{name}\ ;$$

$\mathcal{H}[o, alloc]$ says that $o$ is allocated in $\mathcal{H}$.

Not all mathematical maps are heaps reachable in a Spec$^\flat$ program. We introduce a function $wellFormed(h)$ to describe that $h$ is a reachable heap.

$$\mathbf{function}\ wellFormed([\mathbf{ref}, \mathbf{name}]\mathbf{any})\ \mathbf{returns}\ (\mathbf{bool})\ ;$$

In a well-formed heap, reference-valued fields map allocated references to allocated references of the appropriate type.

$$\begin{array}{l} \mathbf{axiom}\ (\forall\, h\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any} \bullet\ wellFormed(h)\ \Rightarrow \\ \quad (\forall\, r\colon \mathbf{ref},\ f\colon \mathbf{name} \bullet\ r \neq \mathbf{null} \wedge h[r, alloc] \wedge fieldType(f) <: object\ \Rightarrow \\ \quad\quad h[r,f] = \mathbf{null} \vee (h[h[r,f], alloc] \wedge type(h[r,f]) <: fieldType(f))\ ))\ ; \end{array}$$

We introduce a function that relates the heap at two successive program points. It says that the new heap is well-formed and that every reference allocated in the old heap is also allocated in the new heap.

$$\begin{array}{l} \mathbf{function}\ successor([\mathbf{ref}, \mathbf{name}]\mathbf{any},\ [\mathbf{ref}, \mathbf{name}]\mathbf{any})\ \mathbf{returns}\ (\mathbf{bool})\ ; \\ \mathbf{axiom}\ (\forall\, _old\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any},\ _new\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any} \bullet \\ \quad successor(_old, _new)\ \Rightarrow \\ \quad\quad wellFormed(_new) \wedge \\ \quad\quad (\forall\, r\colon \mathbf{ref} \bullet\ _old[r, alloc]\ \Rightarrow\ _new[r, alloc]\ )) \end{array}$$

The elements of an array are stored as one “big” value in a ghost field called $elems$:

$$\mathbf{const}\ elems\colon \mathbf{name}\ ;$$

For example, the Spec$^\flat$ array dereference expression $a[j]$ is translated into BoogiePL$^\flat$ as $Select(\mathcal{H}[a, elems],\ j)$. The length of an array is modeled as a function:

$$\mathbf{function}\ length(\mathbf{ref})\ \mathbf{returns}\ (\mathbf{int})\ ;$$

Array elements are assigned and updated using the following functions.

$$\begin{array}{l} \mathbf{function}\ Select(\mathbf{any}, \mathbf{int})\ \mathbf{returns}\ (\mathbf{any})\ ; \\ \mathbf{function}\ Store(\mathbf{any}, \mathbf{int}, \mathbf{any})\ \mathbf{returns}\ (\mathbf{any})\ ; \end{array}$$

These functions are related as follows:

$$\begin{array}{l} \mathbf{axiom}\ (\forall\, e\colon \mathbf{any},\ i\colon \mathbf{int},\ j\colon \mathbf{int},\ v\colon \mathbf{any} \bullet \\ \quad (i = j\ \Rightarrow\ Select(Store(e, i, v), j) = v) \wedge \\ \quad (i \neq j\ \Rightarrow\ Select(Store(e, i, v), j) = Select(e, j))\ )\ ; \end{array}$$

In a well-formed heap, reference values stored in $elems$ fields are allocated and of the appropriate type:

$$
\begin{array}{l}
\textbf{axiom}\ (\forall\, h\colon [\textbf{ref}, \textbf{name}]\textbf{any} \bullet\ \mathit{wellFormed}(h)\ \Rightarrow \\
\quad (\forall\, r\colon \textbf{ref},\ i\colon \textbf{int} \bullet \\
\qquad r \neq \textbf{null} \wedge h[r, \mathit{alloc}] \wedge \mathit{elemType}(\mathit{type}(r)) <: \mathit{object} \wedge \\
\qquad 0 \leqslant i \wedge i < \mathit{length}(r) \Rightarrow \\
\qquad\quad \mathit{Select}(h[r, \mathit{elems}], i) = \textbf{null} \vee \\
\qquad\quad (h[\mathit{Select}(h[r, \mathit{elems}], i), \mathit{alloc}] \wedge \\
\qquad\qquad \mathit{type}(\mathit{Select}(h[r, \mathit{elems}], i)) <: \mathit{elemType}(\mathit{type}(r)))\ ))\ ;
\end{array}
$$

*Object constructor* Class **object** is built into the language, so we predefine the specification of its constructor:

$$\textbf{procedure}\ \mathit{object}^\circ\mathit{object}(\mathit{this}\colon \textbf{ref})\ \textbf{returns}\ (\,)\ ;$$

### 2.3.1. Classes and Fields

In the sequel, we think of the translation as producing a stream of BoogiePL$^\flat$ program text. The translation is described formally using the function $Tr$, which takes a Spec$^\flat$ fragment and produces a BoogiePL$^\flat$ fragment.

For the translation of a *program*, which consists of a list classes, we have:

$$
\begin{array}{l}
Tr[\![\mathit{classes}]\!] = \\
\quad \text{for each } c \in \mathit{classes} \text{ do} \\
\qquad Tr[\![c]\!]
\end{array}
$$

The prescription of the translation requires control structures, which we introduce as meta-syntax, such as the "for each ... do..." construct here. Note that meta-syntax is written in a Roman font.

We translate a *class declaration* as follows:

$$
\begin{array}{l}
Tr[\![\textbf{class}\ T : S\ \{\ \mathit{members}\ \}]\!] = \\
\quad \textbf{const}\ T\colon \textbf{name}\ ; \\
\quad \textbf{axiom}\ \mathit{superclass}(T) = S\ ; \\
\quad \text{for each } m \in \mathit{members} \text{ do} \\
\qquad Tr[\![m]\!]
\end{array}
$$

For brevity, we use a star to map a translation function over a list of fragments. In this notation, we write the last two lines as just:

$$Tr^*[\![\mathit{members}]\!]$$

As usual, we are sloppy with the connectives between the translated fragments; the implicit connectives are conjunction, some punctuation, or white space.

In the rest of this subsection, we use $\mathbb{C}$ to denote the name of the current class.

*Field declarations* are translated as follows:

$$
\begin{array}{l}
Tr[\![T\ f;]\!] = \\
\quad \textbf{const}\ \mathbb{C}^\circ f\colon \textbf{name}\ ; \\
\quad \textbf{axiom}\ \mathit{fieldType}(\mathbb{C}^\circ f) = \mathit{Type}[\![T]\!]\ ;
\end{array}
$$

We use "$^\circ$" as just another character that can appear as part of identifier names in BoogiePL$^\flat$, but that cannot be used in Spec$^\flat$. Translation function *Type* gives the BoogiePL$^\flat$ term for Spec$^\flat$ types:

$$\begin{array}{lcll} \mathit{Type}[\![\mathbf{bool}]\!] & = & _\mathit{bool} & \\ \mathit{Type}[\![\mathbf{int}]\!] & = & _\mathit{int} & \\ \mathit{Type}[\![T]\!] & = & T & \text{for any object type } T \\ \mathit{Type}[\![T[\,]]\!] & = & \mathit{array}(\mathit{Type}[\![T]\!]) & \end{array}$$

### 2.3.2. Methods

The translation of *method declarations* is more involved. Recall that BoogiePL$^\flat$ only has procedures, no instance methods, so we add **this** as an explicit parameter to the generated procedure. Furthermore, since BoogiePL$^\flat$ types are semantic-less, we instead preserve Spec$^\flat$ types via specifications. Also, BoogiePL$^\flat$ has no built-in notion of heap properties, so we preserve properties like allocatedness of references via specifications. BoogiePL$^\flat$ has no notion of inheritance, so we translate overriding and strengthening of postconditions using multiple procedures. Finally, BoogiePL$^\flat$ syntactically distinguishes calls with possible side effect from side-effect free expressions; thus, we flatten Spec$^\flat$ method bodies as part of the translation into BoogiePL$^\flat$ expressions and commands.

*New Methods* The declaration of a new non-virtual or virtual method in a class $\mathbb{C}$ is translated into BoogiePL$^\flat$ as follows:

$$\begin{array}{l} \mathit{Tr}[\![\mathbf{virtual}^{?}\ T\ m\ (\mathit{Args})\ \ \mathit{Spec}\ \mathit{Body}]\!] = \\ \quad \mathbf{procedure}\ \mathbb{C}^\circ m\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\mathit{Tr}[\![T\ _\mathit{result}]\!])\ ; \\ \qquad \mathit{Tr}^*[\![\mathit{Spec}]\!] \\ \qquad \mathit{TrMod}[\![\mathit{Spec}]\!] \\ \quad \mathbf{implementation}\ \mathbb{C}^\circ m\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\mathit{Tr}[\![T\ _\mathit{result}]\!]) \\ \qquad \{\ \mathbf{assume}\ \mathit{wellFormed}(\mathcal{H})\ ; \\ \qquad\quad \mathbf{assume}\ \mathit{this} \neq \mathbf{null}\ ; \\ \qquad\quad \mathbf{assume}\ \mathit{TypeConstraint}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!]\ ; \\ \qquad\quad \mathit{Tr}[\![\mathit{Body}]\!] \\ \qquad \} \end{array}$$

where formal parameters are translated as follows:

$$\begin{array}{lcll} \mathit{Tr}[\![\mathbf{bool}\ x]\!] & = & x\colon \mathbf{bool} & \\ \mathit{Tr}[\![\mathbf{int}\ x]\!] & = & x\colon \mathbf{int} & \\ \mathit{Tr}[\![T\ x]\!] & = & x\colon \mathbf{ref} & \text{for any reference type } T \\ \mathit{Tr}[\![\mathbf{void}\ x]\!] & = & & \end{array}$$

The last case is intentionally left blank; it is used only for method return types, and a method with a void return type gives rise to no out-parameter is the translation.

The types of the formal parameters we just described are there only to please BoogiePL$^\flat$. The run-time types guaranteed by the static type system of Spec$^\flat$ give rise to assumptions:

$$
\begin{array}{lll}
\mathit{TypeConstraint}[\![\mathbf{bool}\ x]\!] & = & \\
\mathit{TypeConstraint}[\![\mathbf{int}\ x]\!] & = & \\
\mathit{TypeConstraint}[\![T\ x]\!] & = & \text{for any reference type } T \\
\quad x = \mathbf{null} \lor (\mathcal{H}[x, \mathit{alloc}] \land \mathit{type}(x) <: \mathit{Type}[\![T]\!]) & &
\end{array}
$$

For convenience later, we also define:

$$
\mathit{TypeConstraint}[\![\mathcal{H}]\!] \ =
$$

*Overriding Methods* If a method overrides a virtual method, then any new postconditions are added to those previously declared.

$Tr[\![\mathbf{override}\ T\ m\ (\mathit{Args})\ \ \mathit{Spec}\ \mathit{Body}]\!] =$
  $\mathbf{procedure}\ \mathbb{C}{\circ}m\ (\mathit{Tr}[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\mathit{Tr}[\![T\ _\mathit{result}]\!])\ ;$
    for the "$\mathbf{virtual}\ T\ m\ (\mathit{Args})\ \ \mathit{Spec}'\ \mathit{Body}'$" in a superclass of $\mathbb{C}$ do
      $\mathit{Tr}^*[\![\mathit{Spec}']\!]$
      $\mathit{TrMod}[\![\mathit{Spec}']\!]$
    for each "$\mathbf{override}\ T\ m\ (\mathit{Args})\ \ \mathit{Spec}'\ \mathit{Body}'$" in $\mathbb{C}$ or a superclass thereof do
      $\mathit{Tr}^*[\![\mathit{Spec}']\!]$
  $\mathbf{implementation}\ \mathbb{C}{\circ}m\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\mathit{Tr}[\![T\ _\mathit{result}]\!])$
    $\{\ \mathbf{assume}\ \mathit{wellFormed}(\mathcal{H})\ ;$
      $\mathbf{assume}\ \mathit{this} \neq \mathbf{null}\ ;$
      $\mathbf{assume}\ \mathit{TypeConstraint}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!]\ ;$
      $\mathit{Tr}[\![\mathit{Body}]\!]$
    $\}$

*Constructors* For constructors, we automatically grant the license to modify all fields of the object being constructed. The implementation initializes the fields before translating the given constructor body.

$\mathit{Tr}[\![\mathbb{C}\ (\mathit{Args})\ \mathit{Spec}\ \mathit{Body}]\!] =$
  $\mathbf{procedure}\ \mathbb{C}{\circ}\mathbb{C}\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\,)\ ;$
    $\mathit{Tr}^*[\![\mathit{Spec}]\!]$
    $\mathit{TrMod}[\![\mathbf{modifies\ this}.*;\ \mathit{Spec}]\!]$
  $\mathbf{implementation}\ \mathbb{C}{\circ}\mathbb{C}\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\,)$
    $\{\ \mathbf{assume}\ \mathit{wellFormed}(\mathcal{H})\ ;$
      $\mathbf{assume}\ \mathit{this} \neq \mathbf{null}\ ;$
      $\mathbf{assume}\ \mathit{TypeConstraint}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!]\ ;$
      for each field "$F\ f;$" defined in $\mathbb{C}$ do
        $\mathbf{assume}\ \mathcal{H}[\mathit{this}, \mathbb{C}{\circ}f] = \mathit{Zero}[\![F]\!]\ ;$
      $\mathit{Tr}[\![\mathit{Body}]\!]$
    $\}$

where

$$
\begin{array}{lll}
\mathit{Zero}[\![\mathbf{bool}]\!] & = & \mathbf{false} \\
\mathit{Zero}[\![\mathbf{int}]\!] & = & 0 \\
\mathit{Zero}[\![R]\!] & = & \mathbf{null} \qquad \text{for any reference type } R
\end{array}
$$

*Method Specifications* Translating pre- and postconditions is straightforward:

$$\begin{array}{lcl} Tr[\![\textbf{requires}\ E;]\!] & = & \textbf{requires}\ Df[\![E]\!] \wedge Tr[\![E]\!]\ ; \\ Tr[\![\textbf{modifies}\ W;]\!] & = & \\ Tr[\![\textbf{ensures}\ E;]\!] & = & \textbf{ensures}\ Df[\![E]\!] \wedge Tr[\![E]\!]\ ; \end{array}$$

Here, we have opted for the simple design of putting the burden of establishing the definedness of the precondition on callers and the burden of establishing the definedness of the postcondition on the implementation.

To translate the modifies clauses of a method, we first collect all of them and then add the contribution of the modifies list to the method's postcondition. This is described by the following function:

$$\begin{array}{l} TrMod[\![Spec]\!] = \\ \quad \textbf{modifies}\ \mathcal{H}\ ; \\ \quad \textbf{ensures}\ (\forall\, o\!: \textbf{ref},\ f\!: \textbf{name} \bullet \\ \quad\quad o \neq \textbf{null} \wedge \textbf{old}(\mathcal{H})[o, alloc] \Rightarrow \\ \quad\quad\quad ModAllowed[\![Spec, o, f]\!] \vee \\ \quad\quad\quad \mathcal{H}[o,f] = \textbf{old}(\mathcal{H})[o,f]\ ) \end{array}$$

where $ModAllowed$ generates a disjunction of the translated modifies-clause terms:

$$\begin{array}{l} ModAllowed[\![Spec, o, f]\!] = \\ \quad \text{for each ``}\textbf{modifies}\ W;\text{'' in } Spec \text{ do} \\ \quad\quad \text{for each ``}desig\ suffix\text{'' in } W \text{ do} \\ \quad\quad\quad \text{case } suffix \text{ of} \\ \quad\quad\quad\quad (.g): \quad (o = \textbf{old}(Tr[\![desig]\!]) \wedge f = g) \vee \\ \quad\quad\quad\quad (.*): \quad (o = \textbf{old}(Tr[\![desig]\!])) \vee \\ \quad\quad\quad\quad ([*]): \quad (o = \textbf{old}(Tr[\![desig]\!]) \wedge f = elems) \vee \end{array}$$

### *2.3.3. Statements*

The translation of statements needs a preprocessing step, which we call normalizing.

*Normalization* A Spec$^\flat$ body is in normal form, if (i) it is context extended, *i.e.*, all its names are properly resolved; (ii) local variable declarations appear only at the beginning of a block; and (iii) allocations and non-void returning calls appear only as right-hand sides of the designator form $Id$.

We establish (i) as follows: We add **this** as the target expression to each designator $Id$ that references a field or method in scope. We prefix each application of an $Id$ that denotes a method or field in scope (except the built-in fields $Type$ and $Length$), with the most derived class of its definition. For example, if a class $A$ declares a method $M$, a subclass $B$ overrides $M$, and $C$ is a subclass of $B$ that does not declare a further override, then $c.M(EE)$ where $c$ has static type $C$ is normalized into $c.B{\circ}M(EE)$. We normalize each call of the form $\textbf{base}.M(EE)$ into a call $\textbf{this}.S{\circ}M(EE)$, where $S$ is the most derived class of $M$'s definition among superclasses of the enclosing class. Finally, each call of the form $\textbf{base}(EE)$ is normalized into a call $\textbf{this}.S{\circ}S(EE)$ where $S$ is the immediate superclass of the enclosing class.

We establish (ii) by moving each local variable declaration to the beginning of its immediately enclosing block. Spec$^\flat$'s context conditions guarantee that this preserves the meaning of the program.

We establish (iii) by repeatedly applying the following normalizing transformations:

- Let $S$ be an assignment, call, or return statement that contains an allocation or call subexpression. Then, select the leftmost innermost subexpression $e$ in $S$ that is a non-$Id$ designator, a call expression, an allocation expression, or a quantifier, and is not the entire left-hand side of an assignment or the entire call statement; we write $S[e]$ to single out that occurrence of $e$. If such an $e$ exists, then $S[e]$ is normalized into

$$\{T\ x;\ x =\ e;\ S[x]\}$$

where $x$ is a fresh identifier and $T$ is the type of $e$.

- If the guard expression $E$ of a conditional statement **if** $(E)$ $S$ **else** $T$ contains an allocation or call expression, then the conditional statement is normalized into

$$\{\mathbf{bool}\ x;\ x =\ E;\ \mathbf{if}\ (x)\ S\ \mathbf{else}\ T\}$$

- If the guard expression $E$ of a while loop **while** $(E)$ **invariant** $J;\ \{S\}$ contains an allocation or call expression, then the while loop is normalized into

$$\{\mathbf{bool}\ x;\ x =\ E;\ \mathbf{while}\ (x)\ \mathbf{invariant}\ J;\ \{\{S\}\ x =\ E;\}\}$$

This preserves the meaning of the program, since it reflects Spec$^\flat$'s leftmost-innermost evaluation order.

*Translation* We now define the translation of normalized statements.

The translation of blocks is straightforward: translate each variable declaration followed by the translation of the individual statements.

$$\begin{array}{l} Tr[\![\{typeIds\ stmts\}]\!] = \\ \quad \{\ Tr^*[\![typeIds]\!]\ Tr^*[\![stmts]\!]\ \} \end{array}$$

The translations of assert, assume, and return statements check that everything is defined before the corresponding BoogiePL$^\flat$ command is generated:

$$\begin{array}{lcl} Tr[\![\mathbf{assert}\ E;]\!] & = & \mathbf{assert}\ Df[\![E]\!]\ ;\ \mathbf{assert}\ Tr[\![E]\!] \\ Tr[\![\mathbf{assume}\ E;]\!] & = & \mathbf{assert}\ Df[\![E]\!]\ ;\ \mathbf{assume}\ Tr[\![E]\!] \\ Tr[\![\mathbf{return}\ E;]\!] & = & \mathbf{assert}\ Df[\![E]\!]\ ;\ _result := Tr[\![E]\!] \end{array}$$

The bulk of the remaining translation is translating assignments. Field update is translated as follows:

$$\begin{array}{l} Tr[\![E.f =\ F;]\!] = \\ \quad \mathbf{assert}\ Df[\![E]\!]\ ;\ \mathbf{assert}\ Tr[\![E]\!] \neq \mathbf{null}\ ; \\ \quad \mathbf{assert}\ Df[\![F]\!]\ ; \\ \quad \mathcal{H}[Tr[\![E]\!], f] := Tr[\![F]\!]) \end{array}$$

Array update needs to check that the array is non-null, that the index is within the bounds of the array, and that run-time type of $G$ is a subtype of the element type of the run-time type of the array (that is, we check for co-variance).

$$
\begin{array}{l}
\mathit{Tr}[\![ E[F] = \ G; ]\!] = \\
\quad \textbf{assert } \mathit{Df}[\![E]\!] \ ; \textbf{assert } \mathit{Tr}[\![E]\!] \neq \textbf{null} \ ; \\
\quad \textbf{assert } \mathit{Df}[\![F]\!] \ ; \textbf{assert } 0 \leqslant \mathit{Tr}[\![F]\!] \wedge \mathit{Tr}[\![F]\!] < \mathit{length}(\mathit{Tr}[\![E]\!]) \ ; \\
\quad \textbf{assert } \mathit{Df}[\![G]\!] \ ; \textbf{assert } \mathit{type}(\mathit{Tr}[\![G]\!]) <: \mathit{elemType}(\mathit{type}(\mathit{Tr}[\![E]\!])) \ ; \\
\quad \mathcal{H}[\mathit{Tr}[\![E]\!], \mathit{elems}] := \mathit{Store}(\mathcal{H}[\mathit{Tr}[\![E]\!], \mathit{elems}], \mathit{Tr}[\![F]\!], \mathit{Tr}[\![G]\!])
\end{array}
$$

Since the statements we translate are normalized, there are only four cases of local-variable assignments to consider. When the right-hand side is an allocation of an object, then the assignment is translated as follows, where $o$ and $oldHeap$ denote fresh variables:

$$
\begin{array}{l}
\mathit{Tr}[\![ x = \ \textbf{new } C(EE); ]\!] = \\
\quad \{ \ \textbf{var } o\colon \textbf{ref} \ ; \textbf{var } \mathit{oldHeap}\colon [\textbf{ref}, \textbf{name}]\textbf{any} \ ; \\
\quad\quad \textbf{assume } o \neq \textbf{null} \wedge \mathit{type}(o) = C \ ; \\
\quad\quad \textbf{assume } \neg\mathcal{H}[o, \mathit{alloc}] \ ; \\
\quad\quad \mathcal{H}[o, \mathit{alloc}] := \textbf{true} \ ; \\
\quad\quad \textbf{assert } \mathit{Df}^*[\![EE]\!] \ ; \\
\quad\quad \mathit{oldHeap} := \mathcal{H} \ ; \\
\quad\quad \textbf{call } C^\circ C(o, \mathit{Tr}^*[\![EE]\!]) \ ; \\
\quad\quad \textbf{assume } \mathit{successor}(\mathit{oldHeap}, \mathcal{H}) \ ; \\
\quad\quad x := o \\
\quad \}
\end{array}
$$

This translation picks an arbitrary $o$ with the properties that it is non-null, has the appropriate run-time type, and is not yet allocated. The translation then allocates the object $o$ by setting its $alloc$ field to **true**. Finally, it calls the $C$ constructor, adds the assumption that the post-call heap is a well-formed successor of the pre-call heap, and assigns $o$ to the local variable in the assignment statement.

Array allocation is similar:

$$
\begin{array}{l}
\mathit{Tr}[\![ x = \ \textbf{new } T[E]; ]\!] = \\
\quad \{ \ \textbf{var } o\colon \textbf{ref} \ ; \\
\quad\quad \textbf{assume } o \neq \textbf{null} \wedge \mathit{type}(o) = \mathit{Type}[\![T[\,]\,]\!] \ ; \\
\quad\quad \textbf{assume } \neg\mathcal{H}[o, \mathit{alloc}] \ ; \\
\quad\quad \textbf{assert } \mathit{Df}[\![E]\!] \ ; \textbf{assert } 0 \leqslant \mathit{Tr}[\![E]\!] \ ; \\
\quad\quad \textbf{assume } \mathit{length}(o) = \mathit{Tr}[\![E]\!] \ ; \\
\quad\quad \textbf{assume } (\forall\, i\colon \textbf{int} \bullet \\
\quad\quad\quad 0 \leqslant i \wedge i < \mathit{Tr}[\![E]\!] \Rightarrow \mathit{Select}(\mathcal{H}[o, \mathit{elems}], \ i) = \mathit{Zero}[\![T]\!] \ ) \ ; \\
\quad\quad \mathcal{H}[o, \mathit{alloc}] := \textbf{true} \ ; \\
\quad\quad x := o \\
\quad \}
\end{array}
$$

Here, there are two additional assumptions about the reference $o$: that $o$ has the specified length $E$, which we check to be non-negative, and that the elements of $o$ all have zero-equivalent values.

When the right-hand side is a call to a method $T^\circ m$ with return type $R$, then the assignment is translated as follows:

$$\begin{array}{l} Tr[\![x = \;\; E.T^{\circ}m(EE);]\!] = \\ \quad \{ \;\; \mathbf{var}\; oldHeap\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any} \;; \\ \quad\quad \mathbf{assert}\; Df[\![E]\!] \;;\; \mathbf{assert}\; Tr[\![E]\!] \neq \mathbf{null} \;; \\ \quad\quad \mathbf{assert}\; Df^{*}[\![EE]\!] \;; \\ \quad\quad oldHeap := \mathcal{H} \;; \\ \quad\quad \mathbf{call}\; x := T^{\circ}m(\,Tr^{*}[\![E, EE]\!]) \;; \\ \quad\quad \mathbf{assume}\; successor(oldHeap, \mathcal{H}) \;; \\ \quad\quad \mathbf{assume}\; TypeConstraint[\![R\; x]\!] \\ \quad \} \end{array}$$

The last assumption states properties that are guaranteed by the Spec$^\flat$ type system.

For all other assignments to local variables, the translation is:

$$\begin{array}{l} Tr[\![x = \;\; E;]\!] = \\ \quad \mathbf{assert}\; Df[\![E]\!] \;; \\ \quad x := Tr[\![E]\!] \end{array}$$

Call statements are like the calls in local-variable assignments, but they use void methods and have no result value:

$$\begin{array}{l} Tr[\![E.T^{\circ}m(EE);]\!] = \\ \quad \{ \;\; \mathbf{var}\; oldHeap\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any} \;; \\ \quad\quad \mathbf{assert}\; Df[\![E]\!] \;;\; \mathbf{assert}\; Tr[\![E]\!] \neq \mathbf{null} \;; \\ \quad\quad \mathbf{assert}\; Df^{*}[\![EE]\!] \;; \\ \quad\quad oldHeap := \mathcal{H} \;; \\ \quad\quad \mathbf{call}\; T^{\circ}m(\,Tr^{*}[\![E, EE]\!]) \;; \\ \quad\quad \mathbf{assume}\; successor(oldHeap, \mathcal{H}) \\ \quad \} \end{array}$$

The translation of the conditional statement is the one we showed in Section 1.3.

The translation of the while loop in Fig. 5 is almost like we showed in Section 1.3, but we also assume well-formedness properties of the syntactic targets of the loop, and we check and assume some "modifies clauses" on the loop. In particular, we conjoin the postcondition contribution of the enclosing method's modifies clause to the loop invariant. The strengthened loop invariant makes it possible to prove the method's modifies clause at the end of the implementation body. In the definition in Fig. 5, we use *Spec* to denote the declared specification of the enclosing method, prepended with "$\mathbf{modifies}\; \mathbf{this}.*$;" if the enclosing method is a constructor. Note that the expansion of $LoopMod$ produces a predicate that refers to the heap in three different states: $\mathcal{H}$ refers to the current value of the heap, which in this context means the value of the heap on loop-iteration boundaries; $oldHeap$ refers to the value of the heap upon entry to the loop, before any of its iterations; and $\mathbf{old}(\mathcal{H})$, which occurs in the antecedent and may arise in the expansion of $ModAllowed$, refers to the heap on entry to the enclosing method, which is where the method's modifies clause gets its meaning. Note also that, since $LoopMod$ becomes part of the loop invariant, we should in principle check it on entry to the loop, but since by construction it is idempotent, we can omit the check.

Finally, we define $Md$ to return a list of syntactic targets, each of whose form is either "$\mathcal{H}$" or a type-id pair "$T\;\; x$". From such a list $L$, the translation function $StripTypes[\![L]\!]$ that we used above returns $L$ with the type of each type-id pair removed

$$
\begin{array}{l}
Tr[\![\mathbf{while}\ (E)\ \mathbf{invariant}\ J;\ \{S\}]\!] = \\
\quad \{\ \mathbf{var}\ \mathit{oldHeap}\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any}\ ; \\
\quad\quad \mathit{oldHeap} := \mathcal{H}\ ; \\
\quad\quad \mathbf{assert}\ \mathit{Df}[\![J]\!]\ ;\ \mathbf{assert}\ \mathit{Tr}[\![J]\!]\ ; \\
\quad\quad /\!*\ \mathbf{assert}\ \mathit{LoopMod}[\![\mathit{Spec}, \mathit{oldHeap}]\!]\ ;\ *\!/ \\
\quad\quad \mathbf{havoc}\ \mathit{StripTypes}[\![\mathit{Md}[\![S]\!]]\!]\ ; \\
\quad\quad \mathbf{assume}\ \mathit{TypeConstraint}^*[\![\mathit{Md}[\![S]\!]]\!]\ ; \\
\quad\quad \mathbf{assume}\ \mathit{successor}(\mathit{oldHeap}, \mathcal{H})\ ; \\
\quad\quad \mathbf{assume}\ \mathit{Df}[\![J]\!] \wedge \mathit{Tr}[\![J]\!] \wedge \mathit{LoopMod}[\![\mathit{Spec}, \mathit{oldHeap}]\!]\ ; \\
\quad\quad \mathbf{assert}\ \mathit{Df}[\![E]\!]\ ; \\
\quad\quad \{\ \mathbf{assume}\ \mathit{Tr}[\![E]\!] \\
\quad\quad\quad \mathit{Tr}[\![\{S\}]\!]\ ; \\
\quad\quad\quad \mathbf{assert}\ \mathit{Df}[\![J]\!]\ ;\ \mathbf{assert}\ \mathit{Tr}[\![J]\!]\ ;\ \mathbf{assert}\ \mathit{LoopMod}[\![\mathit{Spec}, \mathit{oldHeap}]\!]\ ; \\
\quad\quad\quad \mathbf{assume\ false} \\
\quad\quad [\,]\ \mathbf{assume}\ \neg \mathit{Tr}[\![E]\!] \\
\quad\quad \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{LoopMod}[\![\mathit{Spec}, \mathit{oldHeap}]\!] = \\
\quad (\forall\, o\colon \mathbf{ref},\ f\colon \mathbf{name} \bullet \\
\quad\quad o \neq \mathbf{null} \wedge \mathbf{old}(\mathcal{H})[o, \mathit{alloc}] \Rightarrow \\
\quad\quad\quad \mathit{ModAllowed}[\![\mathit{Spec}, o, f]\!] \vee \\
\quad\quad\quad \mathcal{H}[o, f] = \mathit{oldHeap}[o, f]\ )
\end{array}
$$

**Figure 5.** The translation of while loops.

(we omit the formal definition). In the following, $E$ denotes an expression other than an allocation or call expression, and $x$ denotes an identifier of a type $X$.

$$
\begin{array}{lcl}
\mathit{Md}[\![\{\mathit{typeIds}\ \mathit{stmts}\}]\!] & = & \mathit{Md}^*[\![\mathit{stmts}]\!] \setminus \mathit{typeIds} \\
\mathit{Md}[\![\mathbf{assert}\ E;]\!] & = & \\
\mathit{Md}[\![\mathbf{assume}\ E;]\!] & = & \\
\mathit{Md}[\![x =\ E;]\!] & = & X\ x \\
\mathit{Md}[\![x =\ E.m(EE);]\!] & = & X\ x,\ \mathcal{H} \\
\mathit{Md}[\![x =\ \mathbf{new}\ C(EE);]\!] & = & X\ x,\ \mathcal{H} \\
\mathit{Md}[\![x =\ \mathbf{new}\ T[E];]\!] & = & X\ x,\ \mathcal{H} \\
\mathit{Md}[\![E.f =\ F;]\!] & = & \mathcal{H} \\
\mathit{Md}[\![E[F] =\ G;]\!] & = & \mathcal{H} \\
\mathit{Md}[\![\mathbf{if}\ (E)\ S\ \mathbf{else}\ T]\!] & = & \mathit{Md}[\![S]\!], \mathit{Md}[\![T]\!] \\
\mathit{Md}[\![\mathbf{while}\ (E)\ \mathbf{invariant}\ J;\ \{S\}]\!] & = & \mathit{Md}[\![\{S\}]\!] \\
\mathit{Md}[\![\mathbf{return}\ E;]\!] & = &
\end{array}
$$

### 2.3.4. Expressions

The well-definedness of an expression $E$ is defined by translation function $\mathit{Df}[\![E]\!]$.

$$
\begin{array}{lcll}
Df[\![E \stackrel{.}{\Longrightarrow} F]\!] & = & Df[\![E]\!] \wedge (Tr[\![E]\!] \Rightarrow Df[\![F]\!]) & \\
Df[\![E \mid\mid F]\!] & = & Df[\![E]\!] \wedge (Tr[\![E]\!] \vee Df[\![F]\!]) & \\
Df[\![E \;\&\&\; F]\!] & = & Df[\![E]\!] \wedge (Tr[\![E]\!] \Rightarrow Df[\![F]\!]) & \\
Df[\![E \otimes F]\!] & = & Df[\![E]\!] \wedge Df[\![F]\!] & \text{with } \otimes \text{ being } +, -, \text{ or } * \\
Df[\![E \otimes F]\!] & = & Df[\![E]\!] \wedge Df[\![F]\!] \wedge Tr[\![F]\!] \neq 0 & \text{with } \otimes \text{ being } / \text{ or } \% \\
Df[\![\neg E]\!] & = & Df[\![E]\!] & \\
Df[\![\mathbf{this}]\!] & = & & \\
Df[\![\mathbf{result}]\!] & = & & \\
Df[\![\lambda]\!] & = & & \text{with } \lambda \text{ being any literal} \\
Df[\![x]\!] & = & & \\
Df[\![E.f]\!] & = & Df[\![E]\!] \wedge Tr[\![E]\!] \neq \mathbf{null} & \\
Df[\![E[F]]\!] & = & Df[\![E]\!] \wedge Tr[\![E]\!] \neq \mathbf{null} \wedge & \\
 & & Df[\![F]\!] \wedge 0 \leqslant Tr[\![F]\!] \wedge Tr[\![F]\!] < length(Tr[\![E]\!]) & \\
Df[\![E \ \mathbf{is}\ T]\!] & = & Df[\![E]\!] & \\
Df[\![(T)E]\!] & = & Df[\![E]\!] \wedge (Tr[\![E]\!] = \mathbf{null} \vee type(Tr[\![E]\!]) <: Type[\![T]\!]) & \\
Df[\![\mathbf{old}(E)]\!] & = & \mathbf{old}(Df[\![E]\!]) & \\
Df[\![\mathbf{fresh}(E)]\!] & = & Df[\![E]\!] &
\end{array}
$$

A Spec$^\flat$ expression $E$ is translated into a corresponding BoogiePL$^\flat$ expression by $Tr[\![E]\!]$. In the following, we use $\oplus$ to denote the BoogiePL$^\flat$ operator corresponding to the Spec$^\flat$ operator $\otimes$; except for short-circuit operators (and type setting differences), $\otimes$ and $\oplus$ are the same.

$$
\begin{array}{lcll}
Tr[\![E \otimes F]\!] & = & Tr[\![E]\!] \oplus Tr[\![F]\!] & \\
Tr[\![\neg E]\!] & = & \neg Tr[\![E]\!] & \\
Tr[\![\mathbf{this}]\!] & = & this & \\
Tr[\![\mathbf{result}]\!] & = & _result & \\
Tr[\![\lambda]\!] & = & \lambda & \\
Tr[\![x]\!] & = & x & \\
Tr[\![E.f]\!] & = & \mathcal{H}[Tr[\![E]\!], f] & \text{with } f \text{ not } Type \text{ or } Length \\
Tr[\![E.Type]\!] & = & type(Tr[\![E]\!]) & \\
Tr[\![E.Length]\!] & = & length(Tr[\![E]\!]) & \\
Tr[\![E[F]]\!] & = & Select([Tr[\![E]\!], elems], Tr[\![F]\!]) & \\
Tr[\![E \ \mathbf{is}\ T]\!] & = & Tr[\![E]\!] \neq \mathbf{null} \wedge type(Tr[\![E]\!]) <: Type[\![T]\!] & \\
Tr[\![(T)E]\!] & = & Tr[\![E]\!] & \\
Tr[\![\mathbf{old}(E)]\!] & = & \mathbf{old}(Tr[\![E]\!]) & \\
Tr[\![\mathbf{fresh}(E)]\!] & = & \neg\mathbf{old}(\mathcal{H})[Tr[\![E]\!], alloc] &
\end{array}
$$

Note that we have special cases for the built-in fields *Type* and *Length*, which return the run-time type of an object and the length of an array, respectively.

### 2.4. Example

The Spec$^\flat$ program in Fig. 6 is translated into the BoogiePL$^\flat$ program in Fig. 7.

### 2.5. Summary

To verify object-oriented programs, one needs to define their semantics. One way to do that, which we have followed here, is to translate them into a simpler language with a

$$
\begin{array}{l}
\mathbf{class}\ C : \mathbf{object}\ \{ \\
\quad \mathbf{int}\ x; \\
\quad C(\mathbf{int}\ y)\ \{\ \mathbf{base}();\ \ x =\ y;\ \} \\
\quad \mathbf{int}\ M(\mathbf{int}\ n) \\
\qquad \mathbf{modifies}\ x; \\
\qquad \mathbf{ensures}\ \mathbf{result} = \mathbf{old}(x); \\
\quad \{\ \mathbf{int}\ r =\ x;\ \ x =\ x/n;\ \ \mathbf{return}\ r;\ \} \\
\}
\end{array}
$$

**Figure 6.** An example Spec$^\flat$ program, declaring a class with an integer field, a constructor, and a method.

precisely defined semantics. The simpler language need not be just logical formulas; in fact, there is evidence that including imperative features, closer to the object-oriented language than to logical formulas, makes the encoding of the semantics easier to understand and to implement [53,4]. In the encoding of Spec$^\flat$ that we have presented in this section, we have decided on a storage model, axiomatized types and declarations, and prescribed the translation of statements and expressions. In this translation, we have addressed issues like behavioral subtyping and partiality of operations.

## 3. Invariants and Ownership

To prove the correctness of a method, it is usually necessary to know that its parameters, including the receiver parameter, reference well-formed data, that is, data that satisfy certain consistency conditions. Many consistency conditions can be described by *object invariants*.

This section introduces object invariant patterns and their proof obligations. Syntactically, an object invariant is declared as a class member:

$$
\begin{array}{lll}
\mathit{Member} & ::= & \ldots \\
 & | & \mathit{Invariant}
\end{array}
$$

The declaration gives an invariant for the enclosing class, written in terms of an arbitrary object denoted by the keyword `this`. We start in Section 3.0 with an example that highlights the central question of where invariants hold. In Section 3.1, we look at *intra-object invariants*, which express semantic constraints on the fields of each object. In Section 3.2, we look at an important form of *inter-object invariants*, which express properties of linked objects, that is, of objects that refer to each other. *Aliasing*, the interaction between object references, complicates the handling of inter-object invariants. We employ an *ownership regime* to control the impact of changes among objects. In Section 3.3, we discuss inheritance and dynamic dispatch.

### 3.0. Where Do Invariants Hold?

The quintessential idea of object invariants is that an object satisfies its invariant whenever no constructor or method of the object is active (*cf.* [59]). With some restrictions, this idea can be realized by checking that the constructor establishes the object invariant and that every non-constructor method of the class preserves the invariant.

```
const C: name ;
axiom superclass(C) = object ;
const C°x: name ;
axiom fieldType(C°x) = _int ;
procedure C°C(this: ref, y: int) returns ( ) ;
  modifies ℋ ;
  ensures (∀ o: ref, f: name • o ≠ null ∧ old(ℋ)[o, alloc] ⇒
    o = old(this) ∨ ℋ[o, f] = old(ℋ)[o, f] ) ;
implementation C°C(this: ref, y: int) returns ( )
  { assume wellFormed(ℋ) ; assume this ≠ null ;
    assume this = null ∨ (ℋ[this, alloc] ∧ type(this) <: C ;
    assume ℋ[this, C°x] = 0 ;
    { var oldHeap: [ref, name]any ;                 /* base(); */
      assert this ≠ null ;
      oldHeap := ℋ ;
      call object°object(this) ;
      assume successor(oldHeap, ℋ)
    } ;
    assert this ≠ null ;                            /* this.x = y; */
    ℋ[this, C°x] := y
  }
procedure C°M(this: ref, n: int) returns (_result: int) ;
  ensures _result = old(ℋ[this, C°x]) ;
  modifies ℋ ;
  ensures (∀ o: ref, f: name • o ≠ null ∧ old(ℋ)[o, alloc] ⇒
    (o = old(this) ∧ f = C°x) ∨ ℋ[o, f] = old(ℋ)[o, f] ) ;
implementation C°M(this: ref, n: int) returns (_result: int)
  { assume wellFormed(ℋ) ; assume this ≠ null ;
    assume this = null ∨ (ℋ[this, alloc] ∧ type(this) <: C) ;
    { var r: int ;                                  /* int r; */
      assert this ≠ null ;                          /* r = this.x; */
      r := ℋ[this, C°x] ;
      assert this ≠ null ;                          /* this.x = this.x/n; */
      assert this ≠ null ∧ n ≠ 0 ;
      ℋ[this, C°x] := ℋ[this, C°x]/n ;
      _result := r                                  /* return r; */
    }
  }
```

**Figure 7.** The BoogiePL$^\flat$ translation of the Spec$^\flat$ program in Fig. 6. This figure omits the prelude of the translation, which is described in Section 2.3.0 the same for all translated Spec$^\flat$ programs.

Class *Subject* in Fig. 8 illustrates the idea. Its invariant constrains *st* to be non-zero. The constructor sets *st* to 1, establishing the invariant, and the other methods leave *st* with a non-zero value, maintaining the invariant. Given that **this** satisfies its object invariant on entry to method *Get*, one can prove that the division in the body of *Get* is defined.

```
class Subject {
  Observer obs;
  int st;
  invariant st ≠ 0;

  Subject(Observer o)
  { st = 1; obs = o; }

  void Update(int y)
    requires y ≠ 0;
    modifies this.*, obs.*;
  { st = 0;
    if (obs ≠ null)
      obs.Notify(this);
    st = y;
  }

  int Get()
  { return 1000/st; }
}
```

```
class Observer {
  int cache;

  void Notify(Subject s)
    requires s ≠ null;
    modifies cache;
  { cache = s.Get(); }
}
```

```
class Program {
  void Main() {
    Observer o = new Observer();
    Subject s = new Subject(o);
    s.Update(5);
  }
}
```

**Figure 8.** An example program showing the interaction between two objects, a subject and an observer. Without the invariant declaration, the formalization of Spec$^\flat$ in Section 2 will report one error in this program, namely a division-by-zero error in method *Get*. The **invariant** in *Subject* declares an intention to keep *st* non-zero, but exactly when is the invariant supposed to hold?

The interaction between class *Subject* and class *Observer* illustrates a problem with the basic realization of the quintessential idea of object invariants. Before it calls *Notify* on the observer, method *Update* changes *st* to 0, temporarily violating the invariant. But *Notify* then causes control to *reenter* the subject, which leads to a division-by-zero error. Evidently, there is more to verifying and using object invariants than checking them at the end of methods.

### 3.1. Intra-object Invariants

We introduce a programming discipline, a specification and verification *methodology*, that makes it possible describe and enforce the program's intended design regarding reentrancy and, more generally, regarding object invariants. The methodology explicitly keeps track of when an invariant is known to hold [3].

For every object and array, we introduce a ghost field *inv*, which can take on the values *valid* and *mutable*. The intended meaning of these two states is that the object invariant holds of objects in the *valid* state, but may or may not hold of objects in the *mutable* state. Newly allocated objects are mutable; that is, on entry to a constructor, the object to be constructed is mutable.

For any class $T$ and object $o$ of class $T$, we let $Inv_T[\![o]\!]$ denote the invariant declared in class $T$ applied to object $o$. For an array type $T$, $Inv_T[\![o]\!]$ is just **true**.

In order to be able to do modular verification with object invariants, it is necessary to restrict what an object invariant can depend on. For intra-object invariants, we say an

invariant (declaration) is *admissible* if it only refers to fields of the object, that is, if each of its field-select subexpressions are of the form **this**.*f* for some field *f* (where, as usual, "**this**." can be implicit).

We can now formalize the connection between the $inv$ field and admissible invariants:

**Program Invariant 0** *If the invariant a class $C$ is admissible, then*

$$(\forall\, o \bullet\ o.inv = valid \ \Rightarrow\ Inv_C[\![o]\!]\,)$$

*where the quantification ranges over non-null, allocated objects of type $C$, is a* program invariant*, that is, it holds in every reachable program state.*

To ensure this program invariant, the methodology restricts updates of $inv$ (which occurs in the antecedent of the quantified formula) and updates of other fields of the object (which may occur in the consequent).

Updates of $inv$ are restricted to two new operations, written **unpack** $o$; and **pack** $o$; for any object-valued expression $o$. The idea is that these operations delineate where an object is mutable: **unpack** $o$; makes $o$ mutable, and **pack** $o$; makes $o$ valid after first checking that $Inv_C[\![o]\!]$ holds.

Other field updates are restricted to mutable objects only. That is, we introduce $o.inv = mutable$ as a new precondition of each field update statement $o.f\ =\ E$;.

Applying this methodology to the subject-observer example, we change the code in Fig. 8 as shown in Fig. 9.

*Defaults and Shorthands* This methodology for object invariants uses unpack and pack operations to change $inv$, and uses pre- and postconditions to specify the value of $inv$ on method boundaries. As is exemplified in Fig. 9, these operations and specifications tend to be used in a highly stylized fashion: the constructor ends with a pack operation, state changes in other methods are bracketed by a unpack and pack, the constructor postcondition says that the object is valid, and the precondition of non-constructor methods requires the object to be valid. To simplify the program text, we introduce some defaults.

First, we add a **pack this**; operation at the end of every constructor.

Second, we introduce a structured statement **expose** $(o)$ $\{S\}$ to stand for the common sequence

**unpack** $o$; $\{S\}$ **pack** $o$;

where we assume $S$ does not change $o$. In fact, we only add the **expose** statement, not the **unpack** and **pack** operations, to the Spec♭ language syntax:

$$\begin{array}{lcl} \mathit{Stmt} & ::= & \ldots \\ & | & \textbf{expose}\ (\ \mathit{Expr}\ )\ \mathit{Block} \end{array}$$

where the type of the expression must be a non-**object** reference type.

Third, for every reference-valued method parameter $p$, including **this** unless the method is a constructor, we add the default precondition $p.inv = valid$.

Fourth, we add the default postcondition $inv = valid$ to constructors.

Fifth, because it tends to be a more common specification pattern, we change the definition of $o.*$ in modifies clauses to exclude the $inv$ field (*cf.* page 376):

```
class Subject {
  Observer obs;
  int st;
  invariant st ≠ 0;

  Subject(Observer o)
    ensures inv = valid;
  { st = 1; obs = o;
    pack this;
  }

  void Update(int y)
    requires inv = valid;
    requires y ≠ 0;
    modifies this.∗, obs.∗;
  { unpack this;
    st = y;
    pack this;
    if (obs ≠ null)
      obs.Notify(this);
  }

  int Get()
    requires inv = valid;
  { return 1000/st; }
}
```

```
class Observer {
  int cache;

  void Notify(Subject s)
    requires inv = valid;
    requires s ≠ null ∧ s.inv = valid;
    modifies cache;
  { unpack this;
    cache = s.Get();
    pack this;
  }
}
```

**Figure 9.** The *Subject* and *Observer* classes from Fig. 8, but here using *inv*, **unpack**, and **pack**. The methodology also forced us to change the implementation of *Update*, because there is no way to insert unpack and pack operations in Fig. 8 to live up to the three requirements of: (0) *st* can be updated only if the subject is mutable, (1) the precondition of *Notify* requires that the subject be valid, and (2) the *Subject* invariant must hold at the time of a pack operation. One remaining verification problem, which we address in Section 3.2, is how to make sure the observer *obs* is valid when calling *Notify*.

$ModAllowed[\![Spec, o, f]\!] =$
  for each "**modifies** $W$;" in $Spec$ do
    for each "$desig\ suffix$" in $W$ do
      case $suffix$ of
        $(.*)$ : $(o = \mathbf{old}(Tr[\![desig]\!]) \wedge f \neq inv) \vee$
        $\ldots$

Using these defaults and shorthands, we can simplify the *Update* method of the *Subject* class as follows:

$$\begin{array}{l}
\textbf{void } \mathit{Update}(\textbf{int } y) \\
\quad \textbf{requires } y \neq 0; \\
\quad \textbf{modifies } \textbf{this}.*,\ obs.*; \\
\{\ \textbf{expose } (\textbf{this})\ \{\ st = \ y;\ \} \\
\quad \textbf{if } (obs \neq \textbf{null}) \\
\quad\quad obs.\mathit{Notify}(\textbf{this}); \\
\}
\end{array}$$

*Translation* We change the translation to generate proof obligations that guarantee Program Invariant 0.

We start by changing the translation of how objects and references come into being. For objects, we add an assumption in the constructor, saying that the new object starts in a mutable state (*cf.* page 375):

$$\begin{array}{l}
\mathit{Tr}[\![\mathbb{C}\ (\mathit{Args})\ \mathit{Spec}\ \mathit{Body}]\!] = \\
\quad \textbf{procedure } \mathbb{C}°\mathbb{C}\ \ldots \\
\quad \textbf{implementation } \mathbb{C}°\mathbb{C}\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \textbf{returns } (\,) \\
\quad\quad \{\ \textbf{assume } \mathit{wellFormed}(\mathcal{H})\ ; \\
\quad\quad\quad \textbf{assume } \mathit{this} \neq \textbf{null} \wedge \mathcal{H}[\mathit{this}, \mathit{inv}] = \mathit{mutable}\ ; \\
\quad\quad\quad \textbf{assume } \mathit{TypeConstraint}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!]\ ; \\
\quad\quad\quad \text{for each field ``} F\ f\text{;'' defined in } \mathbb{C} \text{ do} \\
\quad\quad\quad\quad \mathcal{H}[\mathit{this}, \mathbb{C}°f] := \mathit{Zero}[\![F]\!]\ ; \\
\quad\quad\quad \mathit{Tr}[\![\mathit{Body}]\!] \\
\quad\quad \}
\end{array}$$

Remember that our defaults and shorthands add a **pack this**; operation at the end of *Body* (during normalization).

For arrays, we make newly allocated arrays appear in the valid state (note that array types themselves do not have any object invariants) (*cf.* page 378):

$$\begin{array}{l}
\mathit{Tr}[\![x = \ \textbf{new } T[E];]\!] = \\
\quad \{\ \textbf{var } o\colon \textbf{ref}\ ; \\
\quad\quad \ldots \\
\quad\quad \mathcal{H}[o, \mathit{inv}] := \mathit{valid}\ ; \mathcal{H}[o, \mathit{alloc}] := \textbf{true}\ ; \\
\quad\quad x := o \\
\quad \}
\end{array}$$

We add an extra precondition to field and array update (*cf.* page 377). For brevity, here and in the rest of this paper, we assume that expressions like $o$, $a$, $i$, and $e$ in the following translations are local variables; in general, we would first apply normalization and then use $\mathit{Df}[\![\cdot]\!]$ and $\mathit{Tr}[\![\cdot]\!]$, as in Section 2.3.3.

$$\begin{array}{l}
\mathit{Tr}[\![o.f = \ e;]\!] = \\
\quad \textbf{assert } o \neq \textbf{null} \wedge \mathcal{H}[o, \mathit{inv}] = \mathit{mutable}\ ; \\
\quad \mathcal{H}[o, f] := e
\end{array}$$

$$
\begin{array}{l}
Tr[\![a[i] = \ e;]\!] = \\
\quad \textbf{assert}\ a \neq \textbf{null} \wedge \mathcal{H}[a, inv] = mutable\ ; \\
\quad \textbf{assert}\ 0 \leqslant i \wedge i < length(a)\ ; \\
\quad \textbf{assert}\ type(e) <: elemType(type(a))\ ; \\
\quad \mathcal{H}[a, elems] := Store(\mathcal{H}[a, elems], i, e)
\end{array}
$$

Finally, we define the unpack and pack operations as follows, where $o$ has static type $T$.

$$
\begin{array}{l}
Tr[\![\textbf{unpack}\ o;]\!] = \\
\quad \textbf{assert}\ o \neq \textbf{null} \wedge \mathcal{H}[o, inv] = valid\ ; \\
\quad \mathcal{H}[o, inv] := mutable
\end{array}
$$

$$
\begin{array}{l}
Tr[\![\textbf{pack}\ o;]\!] = \\
\quad \textbf{assert}\ o \neq \textbf{null} \wedge \mathcal{H}[o, inv] = mutable\ ; \\
\quad \textbf{assert}\ Df[\![Inv(o)]\!] \wedge Tr[\![Inv(o)]\!]\ ; \\
\quad \mathcal{H}[o, inv] := valid
\end{array}
$$

### 3.2. Inter-object Invariants

An object invariant can span several objects. Suppose object $o$ refers to object $c$ in its invariant; then changing $c$ might invalidate the invariant of $o$. There are several strategies for dealing with this situation [61,3,47,7,67,38,60]. In this section, we deal with the common situation where accesses to $c$ are controlled by $o$. We say that $c$ is part of the *representation* of $o$ and that $o$ is the *owner* of $c$. We do not assume that $c$ knows its owner, and thus we handle the important case where $c$ is an instance of a class defined in a library. These object invariants are called *ownership-based invariants*, because they use the ownership structure of the heap in the definition of which invariants are admissible.

Suppose we want to design a priority queue of tasks, implemented via a sorted singly-linked list of nodes. Figure 10 shows a possible implementation. For a proper working of the priority queue, we design the list so that it is strictly increasing, that is, we need the following invariant for class $Node$:

$$\textbf{invariant}\ next \neq \textbf{null}\ \Rightarrow\ prio < next.prio; \qquad (2)$$

But how can we deal with the fact that the modification of one node's priority might break the invariant in the previous node?

We establish a *hierarchical ownership* relationship on objects. We use ownership to control that, outside an object's invariant, the fields of the object can be mentioned only in the invariants of its transitive owners. The methodology also enforces that when an object is mutable, so are its transitive owners. Consequently, when the fields of an object are changed, it can only violate the invariants of owners, but those owners are mutable, which means the owners are in a state when the invariants are allowed to be violated.

For our list example, we let each node own its successor. To follow the methodology, we must then arrange to expose all predecessors before modify a node.

We encode this ownership regime as follows:

- We extend the domain of the $inv$ field to $\{mutable, valid, committed\}$. We say an object is *committed* if its invariant is known to hold and its owner is not mutable.

```
class PriorityQueue {
  Node hd;

  void Insert(object t, int p)
    requires t ≠ null;
    modifies hd;
  { expose (this) {
      if (hd ≠ null)
        hd = hd.Inject(t, p);
      else
        hd =
          new Node(t, p, null);
    }
  }

  void DeleteMin()
    modifies hd;
  { expose (this) {
      if (hd ≠ null)
        hd = hd.next;
    }
  }

  object Min() {
    if (hd ≠ null)
      return hd.task;
    else
      return null;
  }
}
```

```
class Node {
  object task; int prio;
  Node next;

  Node(object t, int p, Node n) {
    task = t;
    prio = p;
    next = n;
  }

  Node Inject(object t, int p)
    modifies next;
  {
    if (p < prio)
      return new Node(t, p, this);
    expose (this) {
      if (next = null)
        next = new Node(t, p, null);
      else
        next = next.Inject(t, p);
    }
    return this;
  }
}
```

**Figure 10.** An implementation of a priority queue. Formal parameter $t$ denotes a task and $p$ denotes a priority. This version of the implementation points out three verification problems. First, the object invariant that nodes of the priority queue are sorted needs to mention more than one object (namely, $\mathbf{this}.next.prio$), and that is not allowed by the admissibility condition in Section 3.1. Second, how do we know that the receiver objects of the two calls to *Inject* are valid? Third, method *Insert* might change *hd* and the *next* field of an unbounded number of *Node* objects, but all of these modifications could not possibly be listed explicitly in the modifies clause of *Insert*.

- We introduce a field modifier **rep**, which specifies that a field refers to a representation object.

  $$FieldModifier \quad ::= \quad \mathbf{rep}$$

  The **rep** modifier is allowed on fields having reference types.

We can now formalize the meaning of **rep** fields, which establishes an additional property:

**Program Invariant 1** *For any* **rep** *field f declared in a class C,*

```
class Subject {                    void Update(int y)
  rep Observer obs;                  requires y ≠ 0;
  int st;                            modifies this.∗, obs.∗;
  invariant st ≠ 0;                { Observer tmp = obs;
                                     expose (this) { st = y; obs = null; }
  Subject(Observer o)                if (tmp ≠ null)
    modifies o.inv;                    tmp.Notify(this);
  { st = 1; obs = o; }               expose (this) { obs = tmp; }
                                   }
  int Get()
  { return 1000/st; }
```

**Figure 11.** The *Subject* class, updated from Fig. 9. Here, the observer is captured by the subject's constructor to claim ownership of it. The *Update* method temporarily disentangles that ownership relation in order to call *Notify* with two valid objects.

$$(\forall\, o \bullet\; o.inv \neq mutable \;\Rightarrow\; o.f = \mathbf{null} \vee o.f.inv \neq mutable\;)$$

*where the quantification ranges over non-null, allocated objects of type $C$, is a program invariant.*

Admissible invariants for our ownership regime are now restricted as follows: An object $o$ may depend only on the fields of $o$ and the fields of objects transitively owned by $o$. We check this restriction syntactically, allowing an invariant to mention a field-select expression $\mathtt{this}.a.b.\cdots.f.x$ only if $a, b, \ldots, f$ are declared to be **rep** fields. The object invariant (2) of our priority queue example in Fig. 10 has this form, and is thus admissible.

Using our refined methodology, we can solve two of the verification problems in Fig. 10. We declare $hd$ and $next$ as **rep** fields, which make invariant declaration (2) admissible. Program Invariant 1 and the new definition of unpack let us call *Inject*, because they establish that the receiver is valid.

Using our refined methodology, we can also solve the last verification problem with the subject-observer example in Fig. 9. The verifiable code is shown in Fig. 11.

The problem in Fig. 9 was that, on entry to *Update*, we know nothing at all about the validity of $obs$, but we need to know that it is valid when we invoke its *Notify* method. We do know that the subject is valid on entry to *Update*. To entangle the validity of the observer with the validity of the subject, we make the former a representation object of the latter, see the **rep** keyword in Fig. 11.

We now need to admit to "capturing" the valid observer parameter $o$ in the *Subject* constructor. We do that by listing $o.inv$ in the modifies clause of that constructor.

Finally, when the observer is owned by the subject, it is not possible to pass both of them as valid objects to *Notify*. Thus, we temporarily disentangle them, as shown in Fig. 11. While this does make the program verify, it is clumsy. A better solution would be to make the subject and observer *peers*, which means they have the same owner. This solution is explained in detail elsewhere [61,47,54].

*Translation* We change the translation of unpack and pack operations to reflect changes in the committed-status of objects (*cf.* page 388). For a local variable $o$ of static type $T$:

$Tr[\![\mathbf{unpack}\ o;]\!] =$
  $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, inv] = valid$ ;
  $\mathcal{H}[o, inv] := mutable$ ;
  for each field "**rep** $F\ f$;" defined in $T$ do
    { $\mathbf{assume}\ \mathcal{H}[o,f] \neq \mathbf{null}$ ; $\mathcal{H}[\mathcal{H}[o,f], inv] := valid$
    [] $\mathbf{assume}\ \mathcal{H}[o,f] = \mathbf{null}$
    }

$Tr[\![\mathbf{pack}\ o;]\!] =$
  $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, inv] = mutable$ ;
  $\mathbf{assert}\ Df[\![Inv_T[\![o]\!]]\!] \wedge Tr[\![Inv_T[\![o]\!]]\!]$ ;
  for each field "**rep** $F\ f$;" defined in $T$ do
    $\mathbf{assert}\ \mathcal{H}[o,f] = \mathbf{null} \vee \mathcal{H}[\mathcal{H}[o,f], inv] = valid$ ;
  for each field "**rep** $F\ f$;" defined in $T$ do
    { $\mathbf{assume}\ \mathcal{H}[o,f] \neq \mathbf{null}$; $\mathcal{H}[\mathcal{H}[o,f], inv] := committed$
    [] $\mathbf{assume}\ \mathcal{H}[o,f] = \mathbf{null}$
    } ;
  $\mathcal{H}[o, inv] := valid$

*Method Framing Revisited* With the meaning of modifies clauses defined in Section 2.3.2, methods *Insert* and *Inject* in Fig. 10 do not verify. That definition insisted on that every non-new object with a modified field be mentioned explicitly in the modifies clause. But that's absurd. We need some form of abstraction in our modifies clauses. Representation objects establish a natural abstraction boundary that we can use.

With ownership, only owners are allowed to have invariants that depend on representation objects. Since representation objects are implementation details, code should not depend on the exact values of committed objects. Therefore, we now state a more relaxed meaning of modifies clauses: committed objects are allowed to be changed without explicitly being mentioned in modifies clauses.

We change the computing of the postcondition for a Spec$^\flat$ modifies clause (*cf.* page 376) by adding another disjunct:

$TrMod[\![Spec]\!] =$
  $\mathbf{modifies}\ \mathcal{H}$ ;
  $\mathbf{ensures}\ (\forall\, o{:}\,\mathbf{ref},\ f{:}\,\mathbf{name} \bullet$
    $o \neq \mathbf{null} \wedge \mathbf{old}(\mathcal{H})[o, alloc] \Rightarrow$
      $ModAllowed[\![Spec, o, f]\!] \vee$
      $\mathcal{H}[x, inv] = committed \vee$
      $\mathcal{H}[o,f] = \mathbf{old}(\mathcal{H})[o,f]$ )

With the relaxed meaning, the *Insert* method in Fig. 10 does not need to mention any *Node* object in its modifies clause. Likewise, method *Inject* does not need to explicitly mention the modifications of its successor nodes. Since **this** is valid on entry, methods must still announce modifications of fields of **this**, which is why we wrote the modifies clauses of *Insert* and *Inject* in Fig. 10 the way we did. This solves the third verification problem in Fig. 10.

### 3.3. Inheritance and Invariants

Inheritance and virtually dispatched calls are key features of object-oriented programming languages. To discuss their verification problems in more detail, let us introduce the concept of a *class frame*. We say: Each subclass defines one class frame, consisting of its instance variables. Applied to our *Cell* example in Fig. 3, we see that a *Cell* has two frames: the **object** frame, and the *Cell* frame. The latter contains *Cell*'s only instance field, $x$. A *BackupCell* also has a third frame, containing *BackupCell*'s only instance field, $b$. Single inheritance thus results in a sequence of frames.

Let us now look at the problems involved in virtual calls. First, we see that virtual calls lead to classical callback scenarios. For instance, let $c$ be a *BackupCell*; then a call $c.IncBy(3)$ first enters its *Cell* frame, which when evaluating **this**.$Get()$ reenters its *Cell* frame; next, it evaluates $Set(\ldots)$, which enters the *BackupCell* frame, which through a base call reenters the *Cell* frame. How can we maintain the invariants in *Cell* and/or *BackupCell* under such dynamic control flow?

For verification of invariants in the context of inheritance, we let each class frame declare its own invariant. The invariants from different frames of an object are enforced separately. An invariant declared in a class $T$ is admissible if every field-select expression has the form **this**.$a.b.\cdots.f.x$ where, as before, $a, b, \ldots, f$ are declared to be **rep** fields, and the first field ($a$ or $x$) is declared in $T$ or a superclass thereof.

Here is an example:

| | |
|---|---|
| **class** *Cell* { | **class** *BackupCell* : *Cell* { |
|   **int** $x$; |   **int** $b$; |
|   **invariant** $0 \leqslant x$; |   **invariant** $b \leqslant x$; |
|   ... |   ... |

The direct superclass frame of *BackupCell*, namely *Cell*, can be viewed as a rep "object", or rep frame, of *BackupCell* objects. This matches the rep model nicely, since there is only one conceptual pointer from the subclass frame to its immediate superclass frame. Note that, just as for rep objects, the invariant of the "owner" *BackupCell* is allowed to mention fields declared in rep frames (*i.e.*, superclasses).

As in the rep model, we require that all calls (or, more precisely, all **expose** operations) on the rep frame go via calls to its owner, *i.e.*, its subclass frame. Consequently, most methods need to be virtual; you override them in each subclass explicitly, and you always use base calls to transfer control into the superclass, if needed. Embedding a base call in **expose** blocks causes an object's frames to be exposed in a stack-like fashion.

We could introduce an *inv* field for each frame, but since virtual calls and expose statements are used in a highly styled fashion, we can use fewer ghost variables by letting the *inv* field of an object refer to the most derived frame of the object that is valid. That is, $o.inv <: T$ means that $o$ is valid for frame $T$ and all its superclass frame. The properties $o.inv = valid$ and $o.inv = mutable$ that we introduced in Section 3.1 are now represented as $o.inv = type(o)$ and $o.inv =$ **object**, respectively, assuming that **object** has no invariant.

An object can become committed only when all its class frames are valid. To encode the committed state of an object, we introduce a fictitious type named *Committed*, which is modeled as a subtype of all types in the program. Thus, $o.inv = Committed$ means the the object $o$ is committed, previously written as $o.inv = committed$.

With our new encoding of $inv$, the following restates the previous Program Invariants 0 and 1 in the context of subclasses:

**Program Invariant 2** *For any class $C$ with an admissible invariant,*

$$( \forall\, o \bullet\ type(o) <: C \wedge o.inv <: C \Rightarrow Inv_C[\![o]\!] )$$

*and for any* **rep** *field $f$ declared in $C$,*

$$( \forall\, o \bullet\ o.inv <: C \Rightarrow o.f = \mathbf{null} \vee o.f.inv = Committed )$$

*where the quantifications range over non-null, allocated objects, are program invariants.*

*Translation* The following definitions of unpack and pack take the new representation into account. Let $T$ be the static type of $o$, and let $S$ be the immediate superclass of $T$ or, if $T$ is an array type, let $S$ be **object**; then (*cf.* page 391):

$Tr[\![\textbf{unpack } o;]\!] =$
    **assert** $o \neq \mathbf{null} \wedge \mathcal{H}[o, inv] = T$ ;
    $\mathcal{H}[o, inv] := S$ ;
    for each field "**rep** $F\ f$;" defined in $T$ do
      { **assume** $\mathcal{H}[o, f] \neq \mathbf{null}$ ; $\mathcal{H}[\mathcal{H}[o, f], inv] := type(\mathcal{H}[o, f])$
      [] **assume** $\mathcal{H}[o, f] = \mathbf{null}$
      }

$Tr[\![\textbf{pack } o;]\!] =$
    **assert** $o \neq \mathbf{null} \wedge \mathcal{H}[o, inv] = S$ ;
    **assert** $Df[\![Inv_T[\![o]\!]]\!] \wedge Tr[\![Inv_T[\![o]\!]]\!]$ ;
    for each field "**rep** $F\ f$;" defined in $T$ do
      **assert** $\mathcal{H}[o, f] = \mathbf{null} \vee \mathcal{H}[\mathcal{H}[o, f], inv] = type(\mathcal{H}[o, f])$ ;
    for each field "**rep** $F\ f$;" defined in $T$ do
      { **assume** $\mathcal{H}[o, f] \neq \mathbf{null}$; $\mathcal{H}[\mathcal{H}[o, f], inv] := Committed$
      [] **assume** $\mathcal{H}[o, f] = \mathbf{null}$
      } ;
    $\mathcal{H}[o, inv] := T$

Let $f$ be a field declared in a class $C$; then field update is redefined as follows (*cf.* page 387):

$Tr[\![o.f = e;]\!] =$
    **assert** $o \neq \mathbf{null} \wedge \neg(\mathcal{H}[o, inv] <: C)$ ;
    $\mathcal{H}[o, f] := e$

and array update is defined as follows:

$Tr[\![a[i] = e;]\!] =$
    **assert** $a \neq \mathbf{null} \wedge \mathcal{H}[a, inv] = object$ ;
    **assert** $0 \leqslant i \wedge i < length(a)$ ;
    **assert** $type(e) <: elemType(type(a))$ ;
    $\mathcal{H}[a, elems] := Store(\mathcal{H}[a, elems], i, e)$

So that it can be used when proving programs, we add Program Invariant 2 as axioms.

$$\begin{array}{l}\textbf{axiom}\ (\forall\, h{:}\,[\textbf{ref},\textbf{name}]\textbf{any},\ o{:}\,\textbf{ref}\ \bullet\\ \quad \mathit{wellFormed}(h) \wedge o \neq \textbf{null} \wedge h[o, \mathit{alloc}] \Rightarrow\\ \quad\quad \mathit{type}(o) <: C \wedge h[o, \mathit{inv}] <: C \Rightarrow \mathit{Df}[\![\mathit{Inv}_C[\![o]\!]]\!] \wedge \mathit{Tr}[\![\mathit{Inv}_C[\![o]\!]]\!]\ )\ ;\end{array}$$

$$\begin{array}{l}\textbf{axiom}\ (\forall\, h{:}\,[\textbf{ref},\textbf{name}]\textbf{any},\ o{:}\,\textbf{ref}\ \bullet\\ \quad \mathit{wellFormed}(h) \wedge o \neq \textbf{null} \wedge h[o, \mathit{alloc}] \Rightarrow\\ \quad\quad h[o, \mathit{inv}] <: C \Rightarrow h[o, f] = \textbf{null} \vee h[h[o, f], \mathit{inv}] = \mathit{Committed}\ )\ ;\end{array}$$

for every class and applicable field. In these axioms, $\mathit{Inv}_C[\![o]\!]$ is expanded to the expression declared to be the invariant of class $C$, with $o$ replacing occurrences of **this**.

Finally, we add an axiom that says that *Committed* is a subtype of all types.

$$\textbf{axiom}\ (\forall\, T{:}\,\textbf{name}\ \bullet\ \mathit{Committed} <: T\ )\ ;$$

*Method Preconditions Revisited* Exposing an object frame by frame introduces another problem: for every class $T$, the definition or override of a virtual method $m$ in class $T$ is going to unpack the object, and therefore it needs the precondition $\mathit{inv} = T$. For example, to verify the example in Fig. 3 using our methodology, the $\mathit{Cell}{\circ}\mathit{Set}$ and $\mathit{BackupCell}{\circ}\mathit{Set}$ method implementations must expose the object for the $\mathit{Cell}$ and $\mathit{BackupCell}$ frames, respectively, before modifying the fields $x$ and $b$. To meet with the preconditions of such **expose** statements, $\mathit{Cell}{\circ}\mathit{Set}$ would need a precondition of $\mathit{inv} = \mathit{Cell}$ and $\mathit{BackupCell}{\circ}\mathit{Set}$ would need a precondition of $\mathit{inv} = \mathit{BackupCell}$, but a virtual method and its overrides cannot arbitrarily change the method precondition! What condition would we check at call sites?

For call sites that invoke the a virtual method $m$ by $\textbf{base}.m$, we can check different preconditions at different call sites, because base calls are statically bound. That is, calling $\textbf{base}.m$ invokes a particular implementation, so we can arrange to verify, at the call site, the particular precondition required by that implementation. For a dynamically dispatched call to $m$, we cannot statically decide which overridden method will be executed, yet we need to verify, at the call site, that the precondition required by the invoked override holds. By demanding that every class override all inherited virtual methods, the condition to be verified at a dynamically dispatched call to $o.m$ is $\mathit{inv} = \mathit{type}(o)$.

To support these scenarios where different implementations of a method need different preconditions, we introduce a *polymorphic invariant level*, written as $\mathit{inv} = \star$:

$$\begin{array}{lll}\mathit{Literal} & ::= & \ldots\\ & | & \star\end{array}$$

where $\star$ can appear only in the specifications of virtual methods, and the type of $\star$ is that of a run-time type. The idea is that the definition of a method $m$ writes $\mathit{inv} = \star$ in its precondition. For an implementation given in a class $T$, $\mathit{inv} = \star$ means $\mathit{inv} = T$, and for a dynamically dispatched call to $o.m$, it means $\mathit{inv} = \mathit{type}(o)$.

Our $\mathit{Cell}$ and $\mathit{BackupCell}$ classes can now be specified, implemented, and verified as shown in Fig. 12. Note that $\mathit{IncBy}$ is a non-virtual method. With its given specification, its implementation cannot directly perform any update, because it cannot do the necessary **expose**. However, the implementation can still call virtual methods that will expose the object and modify its state.

```
class Cell {
  int x;
  invariant 0 ⩽ x;

  Cell(int i)
    ensures inv = Cell;
  { x = i; }

  virtual int Get()
    requires inv = ⋆;
  { return x; }

  virtual void Set(int i)
    requires inv = ⋆;
  { expose (this) { x = i; } }

  void IncBy(int i)
    requires inv = Type;
  { int t = Get(); Set(t + i); }
}
```

```
class BackupCell : Cell {
  int b;
  invariant b ⩽ x;

  BackupCell(int i)
    ensures inv = BackupCell;
  { base(i); b = i; }

  override int Get()
  { int g;
    expose (this)
      { g = base.Get(); }
    return g;
  }

  override void Set(int i)
  { expose (this)
      { b = x; base.Set(i); }
  }

  virtual int GetBackup()
    requires inv = ⋆;
  { return b; }

  virtual void Rollback()
    requires inv = ⋆;
  { x = b; }
}
```

**Figure 12.** The *Cell* and *BackupCell* classes from Fig. 3 with polymorphic invariant levels. For brevity, we omit all other contracts.

*Defaults and Shorthands* To remove the burden that subclasses must override all virtual methods, our normalization will, for any non-overridden inherited method, insert an override whose body exposes the object and calls the base implementation of the method. For example, for a subclass of *Cell* that does not explicitly override methods *Set* and *Get*, normalization will insert:

```
override void Set(int i)
{ expose (this) { base.Set(i); } }
override int Get()
{ int g; expose (this) { g = Get(); } return g; }
```

Since the value of *inv* is now a type, no longer a boolean, we must make some adjustments in our default method specifications (*cf.* the discussion on Defaults and Shorthands in Section 3.1). For a constructor in a class $C$, we use the default postcondition

```
ensures inv = C;
```

For every virtual method, we use the default precondition

$\textbf{requires}\ inv = \star;$

Third, for every reference-valued method parameter $p$, including **this** unless the method is a constructor or virtual method, we add the default precondition

$\textbf{requires}\ p.inv = p.Type;$

*Translation* We adapt the translation to BoogiePL$^\flat$ as follows. For each definition or override of a virtual method with a polymorphic invariant level, we generate two BoogiePL$^\flat$ procedure declarations (*cf.* page 374):

$Tr[\![MethodModifier\ T\ m\ (Args)\ \ Spec\ Body]\!] =$
$\quad \textbf{procedure}\ \mathbb{C}^\circ Virtual^\circ m\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \textbf{returns}\ (Tr[\![T\ _result]\!]);$
$\quad\quad$ as the previous translation into $\mathbb{C}^\circ m$, but replacing $\star$ with $type(\textbf{this})$
$\quad \textbf{procedure}\ \mathbb{C}^\circ m\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \textbf{returns}\ (Tr[\![T\ _result]\!])\ ;$
$\quad\quad$ as before, but replacing $\star$ with $\mathbb{C}$
$\quad \textbf{implementation}\ \mathbb{C}^\circ m\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \textbf{returns}\ (Tr[\![T\ _result]\!])$
$\quad\quad$ as before

For a call $o.m(\ldots)$ where $o$ has static type $T$ and $m$ is a virtual method, we use $T^\circ Virtual^\circ m$ in the translation. For a call $\textbf{base}.m(\ldots)$ in a class whose superclass is $T$, we use $T^\circ m$ in the translation. The implementation is given for $\mathbb{C}^\circ m$. Our translation does not give any implementation to $\mathbb{C}^\circ Virtual^\circ m$; intuitively, this implementation is provided by the runtime system, which performs the dynamic dispatch by a case split over the run-time type of the receiver object (typically implemented by dereferencing the v-table).

### *3.4. Summary*

The verification of programs requires invariants, but, as we have seen, dealing with invariants presents several verification problems. In this section, we have presented a methodology the structures a program and its specifications in such a way that it is possible to perform sound modular verification. The methodology introduces the field modifier **rep**, the ghost field $inv$, the **expose** statement, and the invariant-level literal $\star$. We can now specify and generate verification conditions for programs with reentrancy, subclassing, dynamic dispatch, and invariants that span several objects and class frames.

## 4. Multi-threaded Programs

Multi-threaded object-oriented programs are becoming mainstream: servers are already multi-threaded, but soon we will have multi-cores on every desktop, too. So the question arises: Can we adapt the single-threaded verification methodology to verify multi-threaded programs? In particular, can we maintain invariants and also prevent data races and deadlocks?

Section 4.0 introduces a methodology to avoid data races for individual objects. Section 4.1 extends the methodology to guarantee inter-object invariants over rep objects. Section 4.2 concludes by extending this methodology to protect against deadlocks.

```
class Counter : Runnable {
  int dangerous;
  Counter() {
    dangerous = 0;
  }
  override void Run() {
    int tmp = dangerous;
    dangerous = tmp + 1;
  }
}
```

```
class Program {
  void Main() {
    Counter ct = new Counter();
    Thread t = new Thread(ct);
    Thread u = new Thread(ct);
    t.Start(); u.Start();
  }
}
```

**Figure 13.** A simple program to illustrate the possible effects of race conditions. Method *Run* is invoked twice in this program (via the *Thread°Start* method), but the final value of *dangerous* may end up as either 1 or 2, depending on how the runtime system's thread scheduler happens to interleave the thread executions.

### 4.0. Data Race Prevention

A *data race* occurs in a multi-threaded program when one thread writes a field or array element, another thread reads or writes the same field or array element, and neither thread performs a synchronization operation that would give it exclusive access to the data. Data races almost always indicate a programming error and such errors are extremely difficult to find and debug due to the nondeterministic interleaving of the thread executions.

Figure 13 shows this problem using a straightforward program. An instance of the *Counter* class is shared by two threads. Looking at the *Run* method of the *Counter* object, which is invoked by the *Thread°Start* method, each thread appears to increment the variable by 1. However, in some interleavings of the thread executions, the combined effect is not to increment the variable by 2. In particular, both threads might read the variable when its value is 0, in which case each of the two threads will set the variable to 1.

Like in C$^\sharp$ and Java, every object in Spec$^\flat$ also acts as a lock. These locks can be used to ensure mutual exclusion among threads by using a lock statement:

$$
\begin{array}{lll}
\mathit{Stmt} & ::= & \ldots \\
 & | & \textbf{lock}\ (\ \mathit{Expr}\ )\ \mathit{Block}
\end{array}
$$

where the expression must be of a reference type. The execution of **lock** $(o)$ $\{S\}$ acquires the lock $o$, executes $S$, and then releases $o$. The acquire operation first waits until a time when no thread holds $o$, so that the acquisition of $o$ will maintain the program invariant that each lock is held by at most one thread at a time.

The locking mechanism prevents multiple threads from holding $o$ at the same time, but it does not prevent threads from accessing $o$'s fields. We introduce a methodology where a thread $t$ can access a field of an object $o$ only if $o$ is *thread local*—that is, the thread that created the object has not made the object available to other threads—or $t$ holds the lock $o$. We call the set of objects that a thread can access its *access set*. By making sure that access sets are disjoint, we prevent data races.

The life cycle of each object can now be described as follows.

- A new object is initially thread local (that is, *unshared*), and is included in the access set of the creating thread.

- An unshared object can be made accessible to other threads by sharing it. The sharing operation removes the object from the thread's access set.
- A shared object can be exclusively acquired by locking it. When (and if) the acquisition succeeds, the object is added to the access set of the thread.
- When a locked object is released, it is removed from the access set of the thread and once again becomes available for acquisition.

*Language Constructs* We introduced the lock statement above. Here, we introduce ghost variables and other constructs needed to write and specify multi-threaded programs.

- We introduce a new keyword that denotes the thread object of the current thread:

$$\begin{array}{lcl} Atom & ::= & \ldots \\ & | & \mathbf{tid} \end{array}$$

  The type of **tid** is the predefined class *Thread*, and it evaluates to a different value for each thread.
- For each object and array, we introduce a boolean ghost field *shared*, which indicates if the object or array is shared or thread local. Note that *shared* is monotonic in the sense that once an object becomes shared, it remains shared forever.
- For each object and array, we introduce a ghost field *mythread*, which refers to the thread with access to the object or **null** if the object is free. Thus, the access set of a thread $t$ is the set of objects whose *mythread* field is $t$. We use this encoding of access sets because it provides us with an appropriate and existing mechanism to handle modifications of access sets (see the discussion on Method Framing Revisited in Section 3.2). The only operation allowed on *mythread* is to compare it with **tid**, and *mythread* is not admissible in object invariants.
- We introduce a statement for sharing thread-local objects:

$$\begin{array}{lcl} Stmt & ::= & \ldots \\ & | & ShareStmt \\ ShareStmt & ::= & \mathbf{share}\ Expr\ ; \end{array}$$

  where the expression must be of a reference type.

Finally, in Fig. 14, we give the specifications of the predefined classes *Thread* and *Runnable*. The precondition and modifies clause of the *Thread* constructor say that a *Runnable* object cannot be used with more than one thread. Similarly, the precondition of *Start* and the inclusion of **this**.*mythread* in the modifies clause of *Start* prevent a thread from being started more than once.

*Example* Let us consider a variation of the Fig. 13 introductory counter example, where each thread runs a session object and several session objects share the counter, see Fig. 15.

The main thread creates a new counter, makes it available for sharing, and creates two session objects, $a$ and $b$. At that point, the sessions objects are thread local to the main thread.

Next, the two threads $t$ and $u$ are created. They take $a$ and $b$ as arguments, both of which are still thread local to the main thread. According to its specification, the thread constructor may remove the *Runnable* object from the caller's access set; hence, the main thread cannot access $a$ and $b$ after constructing the threads.

```
class Thread {
  Thread(Runnable r)
    requires r ≠ null && r.mythread = tid && ¬r.shared;
    modifies r.mythread, r.inv;
    ensures mythread = tid && ¬shared;
  void Start()
    requires mythread = tid;
    modifies this.*, mythread;
  ...
}

class Runnable {
  virtual void Run()
    requires mythread = tid;
    modifies this.*;
}
```

**Figure 14.** The predefined classes *Thread* and *Runnable*.

```
/* main thread */
Counter ct = new Counter();
share ct;
Session a = new Session(ct, 0);
Session b = new Session(ct, 1);
Thread t = new Thread(a);
Thread u = new Thread(b);
t.Start();
u.Start();
```

```
class Counter {
  int n;
  Counter()
    ensures mythread = tid &&
              ¬shared;
  { n = 0; }
  virtual void Inc()
    requires mythread = tid;
    modifies this.*;
  { expose (this) { n = n + 1; } }
}
class Session : Runnable {
  Session(Counter ct, int id)
    requires ct ≠ null &&
                ct.shared;
    ensures mythread = tid &&
              ¬shared;
  ...
}
```

**Figure 15.** An example where multiple threads run session objects that use a shared counter object.

Next, threads $t$ and $u$ are started, which implicitly calls the *Run* method on $a$ and $b$, respectively. Each session object's *Run* method is executed with **tid** set to the executing thread; here, $t$ and $u$, respectively.

*Defaults and Shorthands* Like the specifications we saw earlier for single-threaded program, specifications of multi-threaded programs are written in a stylized fashion. To sim-

plify the program text, we introduce the following defaults (applied during normalization). For every constructor, we use the default postcondition

$$\textbf{ensures } \mathit{mythread} = \textbf{tid} \ \&\& \ \neg \mathit{shared};$$

and for every non-constructor method, we use the default precondition

$$\textbf{requires } \mathit{mythread} = \textbf{tid};$$

These defaults have the additional advantage that they always hold in single-threaded programs. Thus, any part of a program that would verify under the single-threaded methodology will also verify under the multi-threaded methodology.

Most methods do not modify *shared* or *mythread*, so, as we have already assumed in examples, we change the definition of $o.*$ in modifies clauses to exclude these fields (*cf.* page 386):

$$\begin{array}{l} \mathit{ModAllowed}[\![\mathit{Spec}, o, f]\!] = \\ \quad \text{for each ``}\textbf{modifies } W\text{;'' in } \mathit{Spec} \text{ do} \\ \quad\quad \text{for each ``}\mathit{desig\ suffix}\text{'' in } W \text{ do} \\ \quad\quad\quad \text{case } \mathit{suffix} \text{ of} \\ \quad\quad\quad\quad (.*): \quad (o = \textbf{old}(\mathit{Tr}[\![\mathit{desig}]\!]) \wedge f \neq \mathit{inv} \wedge \\ \quad\quad\quad\quad\quad\quad\quad f \neq \mathit{shared} \wedge f \neq \mathit{mythread}) \vee \\ \quad\quad\quad\quad \ldots \end{array}$$

If needed, a modifies clause can list these fields explicitly.

The defaults let us omit several of the specifications we showed explicitly in Fig. 15.

*Translation* Since we don't intend to verify the implementation of $\mathit{Thread}^{\circ}\mathit{Start}$, we never need to keep track of more than one value for **tid**. Therefore, we simply encode **tid** as a global constant with an unknown value:

$$\begin{array}{l} \textbf{const } \mathit{tid}\colon \textbf{ref}\ ; \\ \textbf{axiom } \mathit{tid} \neq \textbf{null} \wedge \mathit{type}(\mathit{tid}) = \mathit{Thread}\ ; \end{array}$$

The translation of the expression **tid** is:

$$\begin{array}{lcl} \mathit{Df}[\![\textbf{tid}]\!] & = & \\ \mathit{Tr}[\![\textbf{tid}]\!] & = & \mathit{tid} \end{array}$$

In the translation of constructors, we add the assumption

$$\textbf{assume } \neg \mathcal{H}[\mathit{this}, \mathit{shared}] \wedge \mathcal{H}[\mathit{this}, \mathit{mythread}] = \mathit{tid}\ ;$$

on entry to the implementation declaration of $\mathbb{C}^{\circ}\mathbb{C}$ (*cf.* page 387).

We record the monotonicity of *shared* as part of the definition of heap successors (*cf.* page 372):

$$\begin{array}{l} \textbf{axiom } (\forall\, _\mathit{old}\colon [\textbf{ref}, \textbf{name}]\textbf{any},\ _\mathit{new}\colon [\textbf{ref}, \textbf{name}]\textbf{any} \bullet \\ \quad \mathit{successor}(_\mathit{old}, _\mathit{new}) \Rightarrow \\ \quad\quad \mathit{wellFormed}(_\mathit{new}) \wedge \\ \quad\quad (\forall\, r\colon \textbf{ref} \bullet\ _\mathit{old}[r, \mathit{alloc}] \Rightarrow _\mathit{new}[r, \mathit{alloc}]\ ) \wedge \\ \quad\quad (\forall\, r\colon \textbf{ref} \bullet\ _\mathit{old}[r, \mathit{shared}] \Rightarrow _\mathit{new}[r, \mathit{shared}]\ )) \end{array}$$

We change the definedness of field-access and array-access expressions. For $f$ a field different from $mythread$, we have (*cf.* page 381):

$$\begin{array}{ll} Df[\![E.mythread]\!] & = Df[\![E]\!] \wedge Tr[\![E]\!] \neq \mathbf{null} \\ Df[\![E.f]\!] & = Df[\![E]\!] \wedge Tr[\![E]\!] \neq \mathbf{null} \wedge \mathcal{H}[Tr[\![E]\!], mythread] = tid \\ Df[\![E[F]]\!] & = Df[\![E]\!] \wedge Tr[\![E]\!] \neq \mathbf{null} \wedge \mathcal{H}[Tr[\![E]\!], mythread] = tid \wedge \\ & \quad Df[\![F]\!] \wedge 0 \leqslant Tr[\![F]\!] \wedge Tr[\![F]\!] < length(Tr[\![E]\!]) \end{array}$$

Note that the reading of the $mythread$ field may constitute a race condition, but since the only operation we allow on $mythread$ is comparing it with $\mathbf{tid}$, any such race condition is benign because $o.mythread = \mathbf{tid}$ is a stable condition—only a thread itself changes $mythread$ to or from the value $\mathbf{tid}$. This argument also applies to the translations below.

Finally, we change the translation of various statements. For field and array element updates (*cf.* page 393):

$$\begin{array}{l} Tr[\![o.f = \ e;]\!] = \\ \quad \mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, mythread] = tid \wedge \neg(\mathcal{H}[o, inv] <: C) \ ; \\ \quad \mathcal{H}[o, f] := e \end{array}$$

$$\begin{array}{l} Tr[\![a[i] = \ e;]\!] = \\ \quad \mathbf{assert}\ a \neq \mathbf{null} \wedge \mathcal{H}[a, mythread] = tid \wedge \mathcal{H}[a, inv] = object \ ; \\ \quad \mathbf{assert}\ 0 \leqslant i \wedge i < length(a) \ ; \\ \quad \mathbf{assert}\ type(e) <: elemType(type(a)) \ ; \\ \quad \mathcal{H}[a, elems] := Store(\mathcal{H}[a, elems], i, e) \end{array}$$

For the unpack and pack operations (*cf.* page 393):

$$\begin{array}{l} Tr[\![\mathbf{unpack}\ o;]\!] = \\ \quad \mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, mythread] = tid \wedge \mathcal{H}[o, inv] = \ldots \ ; \\ \quad \ldots \end{array}$$

$$\begin{array}{l} Tr[\![\mathbf{pack}\ o;]\!] = \\ \quad \mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, mythread] = tid \wedge \mathcal{H}[o, inv] = \ldots \ ; \\ \quad \ldots \end{array}$$

We define the **share** statement as follows:

$$\begin{array}{l} Tr[\![\mathbf{share}\ o;]\!] = \\ \quad \mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, mythread] = tid \ ; \\ \quad \mathbf{assert}\ \neg\mathcal{H}[o, shared] \ ; \\ \quad \mathcal{H}[o, shared] := \mathbf{true} \ ; \\ \quad \mathcal{H}[o, mythread] := \mathbf{null} \end{array}$$

The lock statement is more involved:

$$
\begin{array}{l}
Tr[\![\textbf{lock}\ (o)\ \{S\}]\!] = \\
\quad \textbf{assert}\ o \neq \textbf{null} \wedge \mathcal{H}[o, \mathit{shared}]\ ; \\
\quad \textbf{assume}\ \mathcal{H}[o, \mathit{mythread}] \neq \mathit{tid}\ ; \\
\quad \{\ \textbf{var}\ \mathit{oldHeap}\colon [\textbf{ref}, \textbf{name}]\textbf{any}\ ; \\
\qquad \mathit{oldHeap} := \mathcal{H}\ ; \\
\qquad \textbf{havoc}\ \mathcal{H}\ ;\ \textbf{assume}\ \mathit{successor}(\mathit{oldHeap}, \mathcal{H})\ ; \\
\qquad \textbf{assume}\ (\forall\, x\colon \textbf{ref},\ f\colon \textbf{name} \bullet\ x \neq o\ \Rightarrow\ \mathit{oldHeap}[x, f] = \mathcal{H}[x, f]\ )\ ; \\
\qquad \textbf{assume}\ \mathcal{H}[o, \mathit{mythread}] = \textbf{null} \\
\quad \}\ ; \\
\quad \mathcal{H}[o, \mathit{mythread}] := \mathit{tid}\ ; \\
\quad Tr[\![\{S\}]\!]\ ; \\
\quad \mathcal{H}[o, \mathit{mythread}] := \textbf{null}
\end{array}
$$

Note that the precondition of the lock statement reads the field $o.\mathit{shared}$, which may constitute a race condition. However, any such race condition is benign, since if the precondition $o.\mathit{shared}$ holds, then it is also stable (due to the monotonicity of $\mathit{shared}$).

The lock statement is thread non-reentrant, which means that a thread will deadlock if it attempts to lock a lock that it already holds. We deal with deadlocks in Section 4.2; here, we simply assume $o.\mathit{mythread}$ to be different from **tid** on entry to the lock statement, since this simplifies the bookkeeping we do around the translation of the body of the lock statement.

The purpose of the **havoc** construction in the translation of the lock statement is to simulate the possible interleavings of other threads, and in particular to simulate their possible effects on the fields of $o$. The following example illustrates the effect of this **havoc** on the verification. Let $o$ be an object of a class that has an integer field $f$:

```
int x;
lock (o) { x = o.f; }
lock (o) { assert x == o.f; }      /* this assert may fail */
```

This example has no race condition. However, since other threads may acquire $o$ and change $o.f$ between the two lock statements, the assertion may fail. The **havoc** command at the beginning of the translation of the second lock statement causes the verification to "forget" the value of $o.f$ from the first lock statement, which causes the verification of the assert to fail.

The translation says that the executions to be verified are those in which the **havoc** command establishes the assumption $\mathcal{H}[o, \mathit{mythread}] = \textbf{null}$, that is, those where $o$ is not held in the state after the **havoc** command. Intuitively, the command **assume** $\mathcal{H}[o, \mathit{mythread}] = \textbf{null}$ waits until no other thread holds $o$. The assignments $\mathcal{H}[o, \mathit{mythread}] := \mathit{tid}$ and $\mathcal{H}[o, \mathit{mythread}] := \textbf{null}$ simulate the acquiring and releasing of $o$'s lock.

### *4.1. Invariants and Ownership Trees*

We have now protected against race conditions. By itself, freedom from race conditions does not guarantee that a program behaves more correctly. For example, consider again our introductory counter example in Fig. 13. If we wrap a **lock** (**this**) block around the reading of field *dangerous* and wrap another **lock** (**this**) block around the update of

*dangerous*, then we have avoided race conditions, but we still end up with a program that may fail to increment *dangerous* by 2. In this section, we consider the locking of whole data structures, and in particular locking that will maintain object invariants.

To protect invariants by locks, we expand our methodology to guarantee the following property:

**Program Invariant 3** *In a multi-threaded program,*

$$(\forall\, o \bullet\; o.shared \wedge o.mythread = \mathbf{null} \Rightarrow o.inv = type(o)\,)$$

*where* $o$ *quantifies over non-null, allocated objects, is a program invariant.*

This property says that when an object is shared but free, then it is valid. In other words, an object invariant can be violated only when the object is in the access set of some thread. To enforce this program invariant, we must ensure that objects are valid when they become free, which affects the precondition of the **share** operation.

To refine the multi-threaded methodology to ownership trees, we need to consider what to do with committed objects. According to the single-threaded methodology, operating on a committed object $o$ must start with unpacking $o$'s owner, which makes $o$ valid. Thus, it is natural to let the unpack operation add the representation objects to the thread's access set. To avoid race conditions, we must then prevent other threads from gaining access to committed objects. We will do that by disallowing **rep** fields from referring to shared objects. Under this refined methodology, one single lock statement locks an entire ownership tree, protecting all its invariants. Note that in the refined methodology, an "unshared" object can be accessed by different threads, but only if the object is part of an ownership tree whose root is shared—that is, when we previously said "thread local", we might now want to say "ownership-tree local".

**Program Invariant 4** *In a multi-threaded program,*

$$(\forall\, o \bullet\; o.inv = Committed \Rightarrow o.mythread = \mathbf{null}\,)$$

*where* $o$ *quantifies over non-null, allocated objects, is a program invariant.*

An object $o$ can in a non-**rep** field, say $o.f$, hold on to a reference to a shared object. To access the data structure behind $o.f$, one would then first need to lock $o.f$. In order to meet with the precondition of the lock statement, it is necessary to know that $o.f$ is shared. But the conjunct $\mathbf{this}.f.shared$ is admissible in an object invariant only if $f$ is a **rep** field, and we have just disallowed **rep** fields from referring to shared objects. Instead, we introduce another field modifier:

$$\begin{array}{lcl} FieldModifier & ::= & \ldots \\ & | & \mathbf{shared} \end{array}$$

Declaring a field $f$ with **shared** adds the implicit object invariant:

$$\mathbf{this}.f = \mathbf{null} \;||\; \mathbf{this}.f.shared$$

Because *shared* is monotonic, it is sound to dereference $f$ in this way in an invariant even when $f$ is not a representation object.

The implicit invariants of **rep** and **shared** fields guarantee the following property:

**Program Invariant 5** *In a multi-threaded program, for any* **rep** *field $f$ declared in a class $C$,*

$$(\forall\, o \bullet\; \mathit{type}(o) <: C \wedge o.\mathit{inv} <: C \;\Rightarrow\; o.f = \mathbf{null} \vee \neg o.f.\mathit{shared}\;)$$

*and for any* **shared** *field $f$ declared in $C$,*

$$(\forall\, o \bullet\; \mathit{type}(o) <: C \wedge o.\mathit{inv} <: C \;\Rightarrow\; o.f = \mathbf{null} \vee o.f.\mathit{shared}\;)$$

*where the quantifications range over non-null, allocated objects, are program invariants.*

*Translation* We change the translation of unpack and pack to update the $\mathit{mythread}$ field of representation objects. The pack statement also needs to check the implicit invariants that come from **rep** and **shared** fields. Let $T$ be the static type of $o$, and let $S$ be the immediate superclass of $T$ or, if $T$ is an array type, let $S$ be **object**; then (*cf.* page 401):

$Tr[\![\mathbf{unpack}\ o;]\!] =$
  **assert** $o \neq \mathbf{null} \wedge \mathcal{H}[o, \mathit{mythread}] = \mathit{tid} \wedge \mathcal{H}[o, \mathit{inv}] = T$ ;
  $\mathcal{H}[o, \mathit{inv}] := S$ ;
  for each field "**rep** $F\ f$;" defined in $T$ do
    { **assume** $\mathcal{H}[o, f] \neq \mathbf{null}$ ;
      $\mathcal{H}[\mathcal{H}[o, f], \mathit{mythread}] := \mathit{tid}$ ;
      $\mathcal{H}[\mathcal{H}[o, f], \mathit{inv}] := \mathit{type}(\mathcal{H}[o, f])$
    [] **assume** $\mathcal{H}[o, f] = \mathbf{null}$
    }

$Tr[\![\mathbf{pack}\ o;]\!] =$
  **assert** $o \neq \mathbf{null} \wedge \mathcal{H}[o, \mathit{mythread}] = \mathit{tid} \wedge \mathcal{H}[o, \mathit{inv}] = S$ ;
  **assert** $\mathit{Df}[\![\mathit{Inv}_T[\![o]\!]]\!] \wedge \mathit{Tr}[\![\mathit{Inv}_T[\![o]\!]]\!]$ ;
  for each field "**rep** $F\ f$;" defined in $T$ do
    **assert** $\mathcal{H}[o, f] = \mathbf{null} \vee$
      $(\mathcal{H}[\mathcal{H}[o, f], \mathit{mythread}] = \mathit{tid} \wedge \mathcal{H}[\mathcal{H}[o, f], \mathit{inv}] = \mathit{type}(\mathcal{H}[o, f]) \wedge$
      $\neg\mathcal{H}[\mathcal{H}[o, f], \mathit{shared}])$ ;
  for each field "**shared** $F\ f$;" defined in $T$ do
    **assert** $\mathcal{H}[o, f] = \mathbf{null} \vee \mathcal{H}[\mathcal{H}[o, f], \mathit{shared}]$ ;
  for each field "**rep** $F\ f$;" defined in $T$ do
    { **assume** $\mathcal{H}[o, f] \neq \mathbf{null}$;
      $\mathcal{H}[\mathcal{H}[o, f], \mathit{inv}] := \mathit{Committed}$ ;
      $\mathcal{H}[\mathcal{H}[o, f], \mathit{mythread}] := \mathbf{null}$
    [] **assume** $\mathcal{H}[o, f] = \mathbf{null}$
    } ;
  $\mathcal{H}[o, \mathit{inv}] := T$

To maintain Program Invariant 3, we add precondition $o.\mathit{inv} = o.\mathit{Type}$ to the **share** statement (*cf.* page 401):

$$
\begin{array}{l}
\mathit{Tr}[\![\textbf{share}\ o;]\!] = \\
\quad \textbf{assert}\ o \neq \textbf{null} \wedge \mathcal{H}[o, \mathit{mythread}] = \mathit{tid}\ ; \\
\quad \textbf{assert}\ \neg\mathcal{H}[o, \mathit{shared}]\ ; \\
\quad \textbf{assert}\ \mathcal{H}[o, \mathit{inv}] = \mathit{type}(o)\ ; \\
\quad \mathcal{H}[o, \mathit{shared}] := \textbf{true}\ ; \\
\quad \mathcal{H}[o, \mathit{mythread}] := \textbf{null}
\end{array}
$$

When locking, we also have to forget the knowledge about owned objects (*cf.* page 402):

$$
\begin{array}{l}
\mathit{Tr}[\![\textbf{lock}\ (o)\ \{S\}]\!] = \\
\quad \textbf{assert}\ o \neq \textbf{null} \wedge \mathcal{H}[o, \mathit{shared}]\ ; \\
\quad \textbf{assume}\ \mathcal{H}[o, \mathit{mythread}] \neq \mathit{tid}\ ; \\
\quad \{\ \textbf{var}\ \mathit{oldHeap}\colon [\textbf{ref}, \textbf{name}]\textbf{any}\ ; \\
\quad\quad \mathit{oldHeap} := \mathcal{H}\ ; \\
\quad\quad \textbf{havoc}\ \mathcal{H}\ ;\ \textbf{assume}\ \mathit{successor}(\mathit{oldHeap}, \mathcal{H})\ ; \\
\quad\quad \textbf{assume}\ (\forall\, x\colon \textbf{ref},\ f\colon \textbf{name}\ \bullet \\
\quad\quad\quad \mathcal{H}[x, \mathit{mythread}] = \mathit{tid}\ \Rightarrow\ \mathit{oldHeap}[x, f] = \mathcal{H}[x, f]\ )\ ; \\
\quad\quad \textbf{assume}\ \mathcal{H}[o, \mathit{mythread}] = \textbf{null} \\
\quad \} \\
\quad \mathcal{H}[o, \mathit{mythread}] := \mathit{tid}\ ; \\
\quad \mathit{Tr}[\![\{S\}]\!]\ ; \\
\quad \mathcal{H}[o, \mathit{mythread}] := \textbf{null}
\end{array}
$$

Note how we deal with forgetting the knowledge about owned objects. Unlike the previous translation of the lock statement, where we only forgot the fields of the object being locked, we now erase knowledge about the entire program state, except for the state of the objects that are accessible by the current thread. This reflects the recent possible effects of other threads on the ownership tree rooted at $o$. It also erases knowledge of committed objects whose transitive owners *are* held by the current thread. This encoding is convenient, because it lets us write the **havoc** construction without defining exactly which committed objects are reachable from the thread's accessible objects—something that presents a difficulty for automatic theorem provers anyway—and we argue that this erasing is okay, because a program should rely only on the invariants of, not the exact field values of, committed objects (this is analogous to how we encoded the postcondition contribution of modifies clauses, see the discussion on Method Framing Revisited in Section 3.2).

Finally, the translation also encodes Program Invariants 3 and 5 as axioms (but not Program Invariant 4, because we don't need it in verification).

$$
\begin{array}{l}
\textbf{axiom}\ (\forall\, h\colon [\textbf{ref}, \textbf{name}]\textbf{any},\ o\colon \textbf{ref}\ \bullet \\
\quad \mathit{wellFormed}(h) \wedge o \neq \textbf{null} \wedge \mathcal{H}[o, \mathit{alloc}]\ \Rightarrow \\
\quad\quad \mathcal{H}[o, \mathit{shared}] \wedge \mathcal{H}[o, \mathit{mythread}] = \textbf{null}\ \Rightarrow\ \mathcal{H}[o, \mathit{inv}] = \mathit{type}(o)\ )\ ;
\end{array}
$$

For every class $C$:

```
class Session : Runnable {
  shared Counter ct;
  invariant ct ≠ null;
  int id;

  Session(Counter ct, int id)
    requires ct ≠ null;
  { this.ct = ct; this.id = id; }

  override void Run()
  { while (true) {
      lock (ct) { ct.Inc(); }
    }
  }
}
```

**Figure 16.** The full *Session* class for the counter example in Fig. 15. Field *ct* is declared with **shared**, since the session object needs to keep track of the fact that it is okay to lock it.

$$
\begin{array}{l}
\textbf{axiom } (\forall\, h\colon [\textbf{ref}, \textbf{name}]\textbf{any},\ o\colon \textbf{ref} \bullet \\
\quad \mathit{wellFormed}(h) \wedge o \neq \textbf{null} \wedge \mathcal{H}[o, \mathit{alloc}] \Rightarrow \\
\qquad \mathit{type}(o) <: C \wedge \mathcal{H}[o, \mathit{inv}] <: C \Rightarrow \\
\qquad\quad \text{for each field ``}\textbf{rep}\ F\ f\text{;'' defined in } C \text{ do} \\
\qquad\qquad \mathcal{H}[o,f] = \textbf{null} \vee \neg \mathcal{H}[\mathcal{H}[o,f], \mathit{shared}] \\
\qquad\quad \text{for each field ``}\textbf{shared}\ F\ f\text{;'' defined in } C \text{ do} \\
\qquad\qquad \mathcal{H}[o,f] = \textbf{null} \vee \mathcal{H}[\mathcal{H}[o,f], \mathit{shared}] \\
\quad ) \, ;
\end{array}
$$

*Example* Let us continue the example from Fig. 15 by showing the whole *Session* class, see Fig. 16.

It is instructive to take a closer look at the verification of the $\mathit{Session}^\circ\mathit{Run}$ method:

0. On entry, the typing assumption in the translation of method implementations tells us $\mathit{type}(\mathit{this}) <: \mathit{Session}$. Also, the default precondition tells us $\mathit{this}.\mathit{inv} = \mathit{Session}$, which by the reflexivity of $<:$ yields $\mathit{this}.\mathit{inv} <: \mathit{Session}$.
1. The heap, $\mathcal{H}$, is a syntactic target of the loop, since the loop calls a method (*Md*, page 380). What is known about the heap on an arbitrary iteration thus comes from *LoopMod* (page 380), which applies the method's modifies clause to the loop. The modifies clause of *Run*, declared in class *Runnable* in Fig. 14, is $\textbf{this}.*$, which stands for the fields of **this** except *inv*, *shared*, and *mythread*. Thus, every iteration of the loop starts with $\mathcal{H}[\mathit{this}, \mathit{inv}]$ and $\mathcal{H}[\mathit{this}, \mathit{mythread}]$ having the same values as when the loop was first reached.
2. By steps 0 and 1, we conclude that

$$\mathit{type}(\mathit{this}) <: \mathit{Session} \wedge \mathit{this}.\mathit{inv} <: \mathit{Session}$$

   holds on entry to each loop iteration.
3. By step 2 and Program Invariant 2, we conclude that *this* satisfies the *Session* invariant on entry to each loop iteration, namely $\mathcal{H}[\mathit{this}, \mathit{ct}] \neq \textbf{null}$. And by

step 2 and Program Invariant 5, we conclude that the **shared** field $ct$ is shared: $\mathcal{H}[\mathcal{H}[this, ct], shared]$. These properties about $\mathcal{H}[this, ct]$ are what we need to discharge the precondition of the lock statement.

4. By the encoding of the lock statement, and in particular by the conditions assumed after the **havoc** command, we have that the object being locked, $\mathcal{H}[this, ct]$, is free. Since $\mathcal{H}[this, ct]$ is checked to be shared before the **havoc** , the definition of *successor* tells us that it remains shared after the **havoc** . By Program Invariant 3, we thus have that $\mathcal{H}[this, ct]$ is valid, which is the precondition we need to establish for the call to *Inc*.
5. We deduce that $\mathcal{H}[this, ct]$ is unchanged by the call to *Inc*, which among other things means it is still non-null, as follows:
   - The encoding of the modifies clause of *Inc* lets us conclude that the call does not change any field of *this*, provided *this* is not the target of the call ($\mathcal{H}[this, ct]$) and provided *this* is not committed at the time of the call.
   - We deduce the dis-equality $this \neq \mathcal{H}[this, ct]$ from the assumption about $\mathcal{H}[\mathcal{H}[this, ct], mythread]$ at the beginning of the encoding of the lock statement.
   - According to proof step 2, *this* is not committed on entry to the loop. Moreover, lock acquisition does not change fields of objects in the thread's access set, and proof step 1 tells us that $\mathcal{H}[this, mythread] = tid$ holds on entry to the loop. )
6. By the definition of *successor*, which upon return from a call we get to assume relates the old and new heap of the call, we have that $\mathcal{H}[this, ct]$ remains shared after the call to *Inc*.
7. By steps 5 and 6, we are able to discharge the proof obligation associated with the pack operation at the end of the **expose** block.

### *4.2. Deadlock Prevention*

A *deadlock* occurs when there is a nonempty set of threads, each of which waits for a lock held by another thread in the set. Deadlocks are programming errors.

The prototypical example for a deadlock is the dining philosophers problem [17], where $n$ philosophers (the threads) sit at a round table, spending their time eating and thinking. There are $n$ forks available (the shared objects), placed between adjacent philosophers at the table. Eating requires the use of two forks. A philosopher can only pick up one fork at a time (philosopher locks a fork). In this setting, there is a possibility of a deadlock, for example if every philosopher holds a left fork and waits for a right fork.

Deadlocks can be avoided if all shared objects are partially ordered and each thread acquires shared objects in ascending order.

We let a program construct a partial order on shared objects. We make this order available in Spec$^\flat$ programs by introducing an irreflexive operator:

$$\begin{array}{lll} \otimes & ::= & \ldots \\ & | & \sqsubset \end{array}$$

where the operands of $\sqsubset$ must be the keyword **lockbound** (explain shortly) or be of a reference type. Operator $\sqsubset$ has the same binding power as $<$.

We also change the share statement so that one can specify the position of a newly shared object in this order:

$$ShareStmt \quad ::= \quad \textbf{share}\ Expr^* \sqsubset Expr \sqsubset Expr^*\ ;$$

where all expressions must have a reference type. In the statement **share** $LL \sqsubset o \sqsubset UU$;, it is the expression $o$ that is being shared. It is checked to evaluate to a non-null value. The objects specified by $LL$ are lower bounds and the objects specified by $UU$ are upper bounds. For every pair of objects $l$ and $u$ in $LL$ and $UU$, respectively, if both $l$ and $u$ are non-null, then the share statement requires $l \sqsubset u$, which ensures that there is a place for $o$ between $LL$ and $UU$.

Finally, we introduce a keyword **lockbound** that indicates an upper bound on all the locks acquired by the current thread.

$$\begin{array}{lcl} Atom & ::= & \ldots \\ & | & \textbf{lockbound} \end{array}$$

To avoid deadlocks, a precondition of the **lock** $(o)$ statement is that $o$ lies above **lockbound**. Statement **lock** $(o)\ \{S\}$ then sets **lockbound** to $o$ before executing $S$, and restores **lockbound** to the old value of **lockbound** after executing $S$. **lockbound** can be used only as an argument to $\sqsubset$.

*Example* Dijkstra proposed a solution to avoid deadlocks of dining philosophers by ordering all the forks and requiring the philosophers to pick up their respective forks in that order [17]. We show that solution for $n = 3$ in Fig. 17. The forks are named $x$, $y$, and $z$ and the philosophers are named $a$, $b$, and $c$. Philosopher $a$ will pick up fork $x$ before fork $y$, philosopher $b$ will pick up fork $y$ before fork $z$, and philosopher $c$ will pick up fork $x$ before fork $z$. Since all philosophers adhere to the same global fork order, thus creating an asymmetry around the table, we avoid deadlocks.

Due to timing issues, this solution might still suffer from starvation. To avoid that problem, one can for example introduce queues of eating requests, that guarantee equal access to a fork by adjacent philosophers. We do not discuss this solution any further.

*Prelude* We extend the prelude by encoding **lockbound** as a global variable:

$$\textbf{var}\ lockbound \colon \textbf{ref}\ ;$$

We introduce an strict partial order called $LockOrder$ (*i.e.*, the relation $LockOrder$ is irreflexive and transitive).

$$\begin{array}{l} \textbf{function}\ LockOrder(\textbf{ref}, \textbf{ref})\ \textbf{returns}\ (\textbf{bool})\ ; \\ \textbf{axiom}\ (\forall\, o \colon \textbf{ref} \bullet\ \neg LockOrder(o, o)\ )\ ; \\ \textbf{axiom}\ (\forall\, o \colon \textbf{ref},\ p \colon \textbf{ref},\ q \colon \textbf{ref} \bullet \\ \quad LockOrder(o, p) \wedge LockOrder(p, q)\ \Rightarrow\ LockOrder(p, q)\ )\ ; \end{array}$$

Just like we predefined the **object** class and added a constructor to the prelude, we predefine the $Runnable$ class (see Fig. 14). This lets us give the $Run$ method a specification that is not expressible in the Spec$^\flat$ language; in particular, we include a precondition that says that $lockbound$ is below all references in the Spec$^\flat$ program:

```
class Program {
  voidMain() {
    Forkx = new Fork(); share ⊏ x ⊏;
    Forky = new Fork(); share x ⊏ y ⊏;
    Forkz = new Fork(); share y ⊏ z ⊏;
    Philosopher a = new Philosopher(x, y);
    Philosopher b = new Philosopher(y, z);
    Philosopher c = new Philosopher(x, z);
    Thread A = new Thread(a);
    Thread B = new Thread(b);
    Thread C = new Thread(c);
    A.Start(); B.Start(); C.Start();
  }
}

class Fork { }

class Philosopher {
  shared Fork left; shared Fork right;
  invariant left ≠ null && right ≠ null && left ⊏ right;

  Philosopher(Fork left, Fork right)
    requires left ≠ null && left.shared;
    requires right ≠ null && right.shared;
    requires left ⊏ right;
  { this.left = left; this.right = right; }

  override void Run()
  { while (true) {
      lock (right) { lock (left) {
        /* use the forks to eat ... */
    } } }
  }
}
```

**Figure 17.** A program that runs 3 dining philosophers.

$$
\begin{array}{l}
\textbf{procedure } \mathit{Runnable}{\circ}\mathit{Run}(\mathit{this}\colon \textbf{ref}) \textbf{ returns } (\,)\ ; \\
\quad \textbf{requires } (\forall\, o\colon \textbf{ref} \bullet \\
\quad\quad o \neq \textbf{null} \wedge \mathit{type}(o) <: \mathit{object} \Rightarrow \mathit{LockOrder}(\mathit{lockbound}, o)\ )\ ; \\
\quad \ldots
\end{array}
$$

We omit here the constant *Runnable*, its associated type axioms, and the translation of the constructor.

*Translation* The translation of the expression **lockbound** is:

$$
\begin{array}{l}
\mathit{Df}[\![\textbf{lockbound}]\!] = \\
\mathit{Tr}[\![\textbf{lockbound}]\!] = \mathit{lockbound}
\end{array}
$$

Since the lock statement is a structured block statement, the value of **lockbound** on exit from a method is the same as it was on entry. However, **lockbound** can have different values during the execution of a method. We therefore put **lockbound** into the modifies and ensures clause of every procedure in our translation (*cf.* page 391):

$TrMod[\![Spec]\!] =$
  $\mathbf{modifies}\ \mathcal{H}, lockbound\ ;$
  $\mathbf{ensures}\ lockbound = \mathbf{old}(lockbound)\ ;$
  $\mathbf{ensures}\ (\forall\, o\colon \mathbf{ref},\ f\colon \mathbf{name} \bullet\ \ldots\ \mathcal{H}[o,f] = \mathbf{old}(\mathcal{H})[o,f]\ )$

We replace the translation of the previous **share** statement (*cf.* page 405) with the following translation of the new **share** statement:

$Tr[\![\mathbf{share}\ LL \sqsubset o \sqsubset UU;]\!] =$
  $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, mythread] = tid\ ;$
  $\mathbf{assert}\ \neg\mathcal{H}[o, shared]\ ;$
  $\mathbf{assert}\ \mathcal{H}[o, inv] = type(o)\ ;$
  for each expression "$l$" in $LL$ do
    for each expression "$u$" in $UU$ do
      $\mathbf{assert}\ l = \mathbf{null} \vee u = \mathbf{null} \vee LockOrder(l, u)\ ;$
  for each expression "$l$" in $LL$ do
    $\mathbf{assume}\ l = \mathbf{null} \vee LockOrder(l, o)\ ;$
  for each expression "$u$" in $UU$ do
    $\mathbf{assume}\ u = \mathbf{null} \vee LockOrder(o, u)\ ;$
  $\mathcal{H}[o, shared] := \mathbf{true}\ ;$
  $\mathcal{H}[o, mythread] := \mathbf{null}$

Finally, we change the translation of the lock statement to check for possible deadlock violations and to update $lockbound$ (*cf.* page 405):

$Tr[\![\mathbf{lock}\ (o)\ \{S\}]\!] =$
  $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, shared] \wedge LockOrder(lockbound, o)\ ;$
  $\{\ \mathbf{var}\ oldHeap\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any}\ ;$
    $oldHeap := \mathcal{H}\ ;$
    $\mathbf{havoc}\ \mathcal{H}\ ;\ \mathbf{assume}\ successor(oldHeap, \mathcal{H})\ ;$
    $\mathbf{assume}\ (\forall\, x\colon \mathbf{ref},\ f\colon \mathbf{name} \bullet$
      $\mathcal{H}[x, mythread] = tid\ \Rightarrow\ oldHeap[x,f] = \mathcal{H}[x,f]\ )\ ;$
    $\mathbf{assume}\ \mathcal{H}[o, mythread] = \mathbf{null}$
  $\}$
  $\{\ \mathbf{var}\ oldLockbound\colon \mathbf{ref}\ ;$
    $oldLockbound := lockbound\ ;$
    $lockbound := o\ ;\ \mathcal{H}[o, mythread] := tid\ ;$
    $Tr[\![\{S\}]\!]\ ;$
    $\mathcal{H}[o, mythread] := \mathbf{null}\ ;\ lockbound := oldLockbound$
  $\}$

Note, since we now deal with deadlocks, we have removed the assumption

$\mathbf{assume}\ \mathcal{H}[o, mythread] \neq tid\ ;$

which previously was part of our translation of the lock statement—the checked condition **lockbound** $\sqsubset o$ implies that the thread does not already hold $o$.

### 4.3. Summary

In this section, we have extended the single-threaded methodology to multi-threaded code. The basic idea is to limit the interaction between threads, so that reasoning can proceed mostly as for single-threaded code, except at certain synchronization points. We have shown how to maintain objects invariants in a multi-threaded setting. A single lock statement acquires exclusive access to all objects in an ownership tree. A program can decide the degree of sharing in a program by deciding to make fields either **rep** fields or **shared** fields. We prevent deadlocks by allowing a program to incrementally and locally specify a global partial order among the objects in the program. Locking objects in ascending order then prevents deadlocks.

## 5. History and Acknowledgments

*Program-Verifier Architecture* Roots of the program-verifier architecture we have described trace back to ESC/Modula-3 [14], a project spearheaded by Greg Nelson. The ESC/Modula-3 checker translated Modula-3 programs into a form of Dijkstra's guarded commands [18,65], from which it generated verification conditions. The ESC/Java checker [23] refined this approach by more clearly defining two forms of an intermediate language [53]. The BoogiePL intermediate language [11] took two more steps by adding a mathematical part to the language, which previously had been passed directly to the theorem prover, and by adding a parser for the language, which for debugging the verifier has been shown to have great value [4].

Filliâtre has also proposed a generation of verification conditions via an intermediate language based on type theory [20]. This has served as the basis for the tool and intermediate verification language Why [21]. Why is being used as the intermediate language for the Java verifier Krakatoa [57] and the C verifier Caduceus [22].

Rather than using VC generation and first-order logic, a program verifier can encode more of the program into the formulas passed to the theorem prover. This approach is followed, for example, by the KeY tool [1] for JavaCard programs, which uses dynamic logic, and the LOOP [36,58] and Jive [62] verifiers for Java, which use extensions of Hoare logic.

*Translation of Languages and Language Features* We have shown a translation of core object-oriented language features. The use of updatable maps (arrays) to model references goes back to Burstall [8]. Modeling the heap as a 2-dimensional array, as done by Poetzsch-Heffter [68], has the advantage that one can quantify over all field names, as we have done extensively.

Leino's thesis [43] gave a translation of object-oriented source-language features, together with constructs like exceptions, records, and deallocation, into guarded commands, along the lines of what was done in ESC/Modula-3. Ecstatic [44] is a core object-oriented language with a weakest-precondition semantics, and includes axioms that encode types and allocation. The ESC/Java translation of annotated Java into guarded commands and axioms employed a number of encoding tricks aimed at improving the per-

formance of the underlying theorem prover [52]. Boogie uses similar encodings, but in this paper we have avoided such "optimizations" in order to make the presentation more straightforward.

*Efficient Formulas* We defined the semantics of BoogiePL$^\flat$ commands in terms of classical weakest preconditions. However, such a definition gives rise to a lot of redundancy that can lead a theorem prover to unnecessary case splits, which easily can turn into unbearable performance [24]. ESC/Modula-3 and ESC/Java used techniques for reducing this redundancy, which is important for a practical checker [24,46].

Unlike the structured commands of BoogiePL$^\flat$ that we used in this paper, BoogiePL has unstructured goto commands. Boogie uses a redundancy-reducing technique based on weakest preconditions to define the semantics of these unstructured commands [5].

*Quantifiers* Another important consideration in the design of a practical automatic verifier is how to give the theorem prover directives of how to instantiate universally quantified expressions. The SMT solver Simplify [12] calls these directives *triggers*, and getting good results from Simplify requires good use of triggers.

*Specification, Abstraction, and Methodology* The first sound modular verification methodology for a significant subset of a modern object-oriented language was given by Müller in his thesis [61]. The particular methodology we presented for using object invariants in single-threaded [3] and multi-threaded [35,37] programs is based on joint work with our Spec$^\sharp$ colleagues. There are extensions of this methodology to visibility-based invariants [47,7,60], static class invariants [48], iterators [33], pure methods [9,34], model fields [49], and subject-observers structures [54].

We made use of committed objects to get abstraction in modifies clauses. Other approaches have used abstraction dependencies [43,50,61], data groups [45,51], separation logic [67], and dynamic frames [38].

We know the technique of changing the specification for method overrides from the work on Fugue [10], which called such specifications *sliding*. We justified the soundness of dereferencing shared fields in implicit object invariants on the grounds that $shared$ is monotonic, which is an idea further explored for type states [19]. The technique of capturing parameters, which we specify by a modifies clause that mentions $inv$ and/or $shared$, was used in the work on ESC/Modula-3 [13].

## 6. Conclusion

Program verification, although as old as computer science [25], is still one of its grand challenges [31].

This paper developed a verifying compiler for a multi-threaded object-oriented subset of Spec$^\sharp$ [6], here called Spec$^\flat$. Correctness of Spec$^\flat$ programs is specified by types, method specifications, object invariants, field modifiers, ghost state, and new statements. The developed compiler (defined in Sections 2, 3, and 4) takes as input a Spec$^\flat$ program, and generates, via an intermediate language called BoogiePL$^\flat$ (defined in Section 1), first-order verification conditions, which can be processed by an SMT solver. If its proof attempt succeeds, then the program is correct and can be run. If it fails, advice is sought from the user. Experience shows that this is a viable approach. By now, many thousands

of Spec$^\sharp$ lines have been verified, although often only with shallow properties like freedom from raised exceptions.

This paper addresses many challenges of verifying modern multi-threaded object-oriented languages. We have shown how to deal with reentrancy, aliasing, inheritance, representation abstraction, method framing, and multi-threading. We have also shown how to engineer a basic verifier by introducing an intermediate verification language. Much remains to be done: for instance, we have to learn how to verify more object-oriented design patterns [26], different kinds of concurrent code [41], and we have to learn how to verify abstractions [32].

We hope that this paper guides students toward understanding program verification, and we encourage them to build their own verifier. Program verification is a rich and rewarding research field.

## References


[0] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[3] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[4] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

[5] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Michael D. Ernst and Thomas P. Jensen, editors, *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05*, pages 82–87. ACM, September 2005.

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[7] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Seventh International Conference on Mathematics of Program Construction (MPC 2004)*, Lecture Notes in Computer Science, pages 54–84. Springer-Verlag, July 2004.

[8] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 6:23–50, 1971.

[9] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[10] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004—Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, June 2004.

[11] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[12] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[13] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, DEC Systems Research Center, July 1998.

[14] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[15] L. Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, University of California, Berkeley, Berkeley, CA 94720, 1973.

[16] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[17] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971.

[18] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[19] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *Proceedings of International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, July 2003.

[20] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[21] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[22] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[23] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[24] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[25] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[27] Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, 1978.

[28] Donald I. Good, Ralph L. London, and W. W. Bledsoe. An interactive program verification system. In *Proceedings of the international conference on reliable software*, pages 482–492. ACM, 1975.

[29] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[31] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[32] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, USA, 2006.

[33] Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. Iterators revisited: Proof rules and implementation. In *Workshop on Formal Techniques for Java-like Programs (FTfJP 2005)*, July 2005.

[34] Bart Jacobs and Frank Piessens. Verification of programs with inspector methods. In *Workshop on Formal Techniques for Java-like Programs (FTfJP 2006)*, July 2006.

[35] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 137–147. IEEE, September 2005.

[36] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, April 2001.

[37] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming

model for concurrent object-oriented programs. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 420–439. Springer, November 2006.

[38] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.

[39] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[40] James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, Pittsburg, PA 15213, September 1969.

[41] Doug Lea. *Concurrent programming in Java: design principles and patterns*. The Java series. Addison-Wesley, Reading, MA, USA, 1996.

[42] Gary Todd Leavens. *Verifying Object-Oriented Programs that Use Subtypes*. PhD thesis, MIT Laboratory for Computer Science, February 1989. Available as Technical Report MIT/LCS/TR-439.

[43] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Technical Report Caltech-CS-TR-95-03.

[44] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[45] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[46] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, March 2005.

[47] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[48] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer, July 2005.

[49] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, March 2006.

[50] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[51] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 246–257. ACM, May 2002.

[52] K. Rustan M. Leino and James B. Saxe. Java to guarded commands translation. Design note ESCJ 16c, ESC/Java source distribution, August 1998.

[53] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.

[54] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. Manuscript KRML 166, October 2006.

[55] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[56] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.

[57] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–

2):89–106, January–March 2004.

[58] Tiziana Margaria and Wang Yi, editors. *The LOOP compiler for Java and JML*, volume 2031 of *Lecture Notes in Computer Science*. Springer, April 2001.

[59] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

[60] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Invariants for non-hierarchical object structures. In Anamaria Martins Moreira and Leila Ribeiro, editors, *Brazilian Symposium on Formal Methods, SBMF 2006*, pages 233–248. SBC, September 2006.

[61] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

[62] Peter Müller, Jörg Meyer, and Arnd Poetzsch-Heffter. Programming and interface specification language of JIVE—specification and design rationale. Technical Report 223, Fernuniversität Hagen, 1997.

[63] John Nagle and Scott Johnson. Practical program verification: Automatic program proving for real-time embedded systems. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 48–58, January 1983.

[64] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Also available as Technical Report CSL-81-10, Xerox PARC, June 1981.

[65] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[66] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[67] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 247–258. ACM, January 2005.

[68] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.

*© 2007 IOS Press. All rights reserved.*

# Author Index

Alur, R. 1
Bhargavan, K. 87
Broy, M. 22
Chaudhuri, S. 1
Cohen, E. 73
Fournet, C. 87
Giuli, T.J. 341
Gordon, A.D. 87
Hoare, T. 116
Knapp, S. 239
Kupferman, O. 135
Lampson, B. 151
Leino, M. 351
Paul, W. 239
Pnueli, A. 298
Prasad, K.V. 341
Rustan, K. 351
Schulte,W. 351
Tse, S. 87
van Lamsweerde, A. 196

This page intentionally left blank